



Silk Test 17.5

Silk Test Classic Open Agent Help

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
<http://www.microfocus.com>

Copyright © Micro Focus 1992-2016. All rights reserved.

MICRO FOCUS, the Micro Focus logo and Silk Test are trademarks or registered trademarks of Micro Focus IP Development Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

All other marks are the property of their respective owners.

2016-11-02

Contents

Licensing Information	18
Getting Started	19
Automation Under Special Conditions (Missing Peripherals)	19
Silk Test Product Suite	20
Contacting Micro Focus	21
Information Needed by Micro Focus SupportLine	22
What's New in Silk Test Classic	23
Mobile Device Live View	23
Testing Hybrid Mobile Applications	23
Enhanced Cross-Browser Testing	23
Using Devices and Browsers that are Managed by Silk Central	23
Usability Enhancements	23
API Enhancements	24
Technology Updates	24
Microsoft Windows 10 Anniversary Update	24
New Mozilla Firefox Versions	25
Google Chrome Support	25
Android Support	25
Java SWT Support	25
Open Agent	26
How Silk Test Classic Assigns an Agent to a Window Declaration	26
Agent Options	26
Setting the Default Agent	45
Setting the Default Agent Using the Runtime Options Dialog Box	45
Setting the Default Agent Using the Toolbar Icons	46
Connecting to the Default Agent	46
Creating a Script that Uses Both Agents	46
Overview of Record Functionality Available for the Silk Test Agents	47
Setting the Window Timeout Value to Prevent Window Not Found Exceptions	48
Manually Setting the Window Timeout Value	48
Setting the Window Timeout Value in the Agent Options Dialog Box	48
Configuring Open Agent Port Numbers	48
Configuring the Port to Connect to the Information Service	49
Open Agent Port Numbers	49
Stopping the Open Agent After Test Execution	50
Basic Workflow for the Open Agent	51
Creating a New Project	51
Configuring Applications	52
Recording Test Cases for Standard and Web Applications	52
Recording Test Cases for Mobile Applications	53
Running a Test Case	54
Viewing Test Results	55
Migrating from the Classic Agent to the Open Agent	56
Differences for Agent Options Between the Silk Test Agents	56
Differences in Object Recognition Between the Silk Test Agents	57
Differences in the Classes Supported by the Silk Test Agents	59
Differences in the Parameters Supported by the Silk Test Agents	63
Overview of the Methods Supported by the Silk Test Agents	64
SYS Functions Supported by the Open Agent and the Classic Agent	64
Silk Test Classic Projects	66

Storing Project Information	66
Accessing Files Within Your Project	67
Sharing a Project Among a Group	68
Project Explorer	68
Creating a New Project	69
Opening an Existing Project	70
Converting Existing Tests to a Project	71
Using Option Sets in Your Project	71
Editing an Options Set	71
Silk Test Classic File Types	72
Organizing Projects	73
Adding Existing Files to a Project	73
Renaming Your Project	73
Working with Folders in a Project	74
Moving Files Between Projects	75
Removing Files from a Project	76
Turning the Project Explorer View On and Off	76
Viewing Resources Within a Project	76
Packaging a Silk Test Classic Project	76
Emailing a Project	78
Exporting a Project	80
Troubleshooting Projects	81
Files Not Found When Opening Project	81
Silk Test Classic Cannot Load My Project File	81
Silk Test Classic Cannot Save Files to My Project	81
Silk Test Classic Does Not Run	82
My Files No Longer Display In the Recent Files List	82
Cannot Find Items In Classic 4Test	82
Editing the Project Files	82
Project Description Dialog Box	83
Enabling Extensions for Applications Under Test	84
Extensions that Silk Test Classic can Automatically Configure	84
Extensions that Must be Set Manually	85
Extensions on Host and Target Machines	85
Enabling Extensions Automatically Using the Basic Workflow	86
Enabling Extensions on a Host Machine Manually	86
Manually Enabling Extensions on a Target Machine	87
Enabling Extensions for Embedded Browser Applications that Use the Classic Agent	88
Enabling Extensions for HTML Applications (HTAs)	88
Adding a Test Application to the Extension Dialog Boxes	89
Verifying Extension Settings	90
Why Applications do not have Standard Names	90
Duplicating the Settings of a Test Application in Another Test Application	90
Deleting an Application from the Extension Enabler or Extensions Dialog Box	91
Disabling Browser Extensions	91
Comparison of the Extensions Dialog Box and the Extension Enabler Dialog Box	91
Configuring the Browser	92
Setting Agent Options for Web Testing	93
Specifying a Browser for Silk Test Classic to Use in Testing a Web Application	93
Specifying your Default Browser	94
Enable Extensions Dialog Box (Classic Agent)	94
Extension Information Dialog Box	95
Extension Settings Dialog Box (.NET)	95
Extension Settings Dialog Box (Web)	95
Extension Settings Dialog Box (Client/Server)	96
Extension Settings Dialog Box (Java)	96

Understanding the Recovery System for the Open Agent	98
Setting the Recovery System for the Open Agent	99
Base State	99
Modifying the Base State (Open Agent)	100
DefaultBaseState Function	101
Adding Tests that Use the Open Agent to the DefaultBaseState	101
DefaultBaseState and the wDynamicMainWindow Object	104
Flow of Control	104
The Non-Web Recovery Systems Flow of Control	104
How the Non-Web Recovery System Closes Windows	104
How the Non-Web Recovery System Starts the Application	105
Modifying the Default Recovery System	105
Overriding the Default Recovery System	106
Handling Login Windows	107
Specifying Windows to be Left Open for Tests that Use the Open Agent	108
Specifying New Window Closing Procedures	108
Specifying Buttons, Keys, and Menus that Close Windows	109
Recording a Close Method for Tests that Use the Open Agent	109
Set Recovery System Dialog Box	110
Test Plans	111
Structure of a Test Plan	111
Overview of Test Plan Templates	112
Example Outline for Word Search Feature	112
Converting a Results File to a Test Plan	114
Working with Test Plans	114
Creating a New Test Plan	114
Indent and Change Levels in an Outline	115
Adding Comments to Test Plan Results	115
Documenting Manual Tests in the Test Plan	116
Describing the State of a Manual Test	116
Inserting a Template	116
Changing Colors in a Test Plan	117
Linking the Test Plan to Scripts and Test Cases	117
Insert Testplan Template Dialog Box	118
Working with Large Test Plans	118
Determining Where Values are Defined in a Large Test Plan	118
Dividing a Test Plan into a Master Plan and Sub-Plans	118
Creating a Sub-Plan	119
Copying a Sub-Plan	119
Opening a Sub-Plan	119
Connecting a Sub-Plan with a Master Plan	119
Refreshing a Local Copy of a Sub-Plan	119
Sharing a Test Plan Initialization File	120
Saving Changes	120
Overview of Locks	120
Acquiring and Releasing a Lock	120
Generating a Test Plan Completion Report	120
Testplan Completion Report Dialog Box	121
Adding Data to a Test Plan	121
Specifying Unique and Shared Data	122
Adding Comments in the Test Plan Editor	122
Testplan Editor Statements	122
The # Operator in the Testplan Editor	122
Using the Testplan Detail Dialog Box to Enter the testdata Statement	122
Entering the testdata Statement Manually	123
Linking Test Plans	123

Linking a Description to a Script or Test Case using the Testplan Detail Dialog Box	123
Linking a Test Plan to a Data-Driven Test Case	124
Linking to a Test Plan Manually	124
Linking a Test Case or Script to a Test Plan using the Testplan Detail Dialog Box	124
Linking the Test Plan to Scripts and Test Cases	124
Example of Linking a Test Plan to a Test Case	125
Categorizing and Marking Test Plans	125
Marking a Test Plan	125
How the Marking Commands Interact	126
Marking One or More Tests	126
Printing Marked Tests	126
Mark By Query Dialog Box	126
Mark By Named Query Dialog Box	128
Using Symbols	128
Overview of Symbols	128
Symbol Definition Statements in the Test Plan Editor	130
Defining Symbols in the Testplan Detail Dialog box	130
Assigning a Value to a Symbol	131
Specifying Symbols as Arguments when Entering a testcase Statement	131
Attributes and Values	131
Overview of Attributes and Values	131
Predefined Attributes	132
User Defined Attributes	132
Adding or Removing Members of a Set Attribute	132
Rules for Using + and -	133
Defining an Attribute and the Values of the Attribute	133
Assigning Attributes and Values to a Test Plan	133
Assigning an Attribute from the Testplan Detail Dialog Box	134
Modifying the Definition of an Attribute	134
Queries	135
Overview of Test Plan Queries	135
Overview of Combining Queries to Create a New Query	135
Guidelines for Including Symbols in a Query	136
The Differences between Query and Named Query Commands	136
Creating a New Query	136
Edit a Query	137
Delete a Query	137
Combining Queries	138
Combine Testplan Queries Dialog Box	138
New/Edit Testplan Query Dialog Box	138
Create Session Dialog Box	139
Testplan Detail Dialog Box	140
Testplan Detail - Testcase Dialog Box	141
Define Attributes Dialog Box	141
New Attribute Dialog Box	142
Update Manual Tests Dialog Box	143
Chart Settings Dialog Box	143
Duplicate Test Descriptions dialog box	144
Designing and Recording Test Cases with the Open Agent	145
Creating Test Cases with the Open Agent	145
Recording Test Cases for Standard and Web Applications	145
Recording Test Cases for Mobile Applications	146
Recording a Test on Microsoft Edge	147
Recording Window Declarations that Include Locator Keywords	148

Recording Locators Using the Locator Spy	149
Recording an Application State	150
Recording Additional Actions Into an Existing Test	151
Actions Available During Recording	151
Setting Recording Options for the Open Agent	152
Best Practices for Creating Test Scripts	153
Specifying Whether to Use Locators or Tags to Resolve Window Declarations ...	154
Saving a Script File	154
Testing an Application State	154
Editing Remote Locations	155
Locator Spy Dialog Box	156
Record Testcase Dialog Box	156
Update Files Dialog Box	157
Dynamic Object Recognition	157
XPath Basic Concepts	158
Supported XPath Subset	159
XPath Samples	161
Supported Locator Attributes	162
Using Locators	162
Using Locators to Check if an Object Exists	162
Identifying Multiple Objects with One Locator	163
Locator Customization	163
Troubleshooting Performance Issues for XPath	168
Highlighting Objects During Recording	169
Overview of the Locator Keyword	169
Test Cases	172
Overview of Test Cases	172
Anatomy of a Basic Test Case	173
Types of Test Cases	173
Test Case Design	173
Constructing a Test Case	174
Data in Test Cases	175
Saving Test Cases	176
Recording Without Window Declarations	176
Overview of Application States	176
Behavior of an Application State Based on NONE	177
Example: A Feature of a Word Processor	177
Verification	179
Verifying Object Properties	179
Overview of Verifying Bitmaps	180
Overview of Verifying an Objects State	181
Fuzzy Verification	182
Verifying that a Window or Control is No Longer Displayed	183
Data-Driven Test Cases	184
Data-Driven Workflow	184
Working with Data-Driven Test Cases	185
Code Automatically Generated by Silk Test Classic	185
Tips And Tricks for Data-Driven Test Cases	186
Testing an Application with Invalid Data	188
Enabling and Disabling Workflow Bars	188
Data Source for Data-Driven Test Cases	189
Creating the Data-Driven Test Case	190
Select Data Source Dialog Box	197
Setup Data Driven Script DSN Dialog Box	198
Specify Data Driven Script Dialog Box	198
Specify Data Driven Testcase Dialog Box	199

Specify Rows Dialog Box	199
Go to Testcase Dialog Box	201
Select Testcase Dialog Box	201
Characters Excluded from Recording and Replaying	202
Testing in Your Environment with the Open Agent	203
Distributed Testing with the Open Agent	203
Remote Testing with the Open Agent	203
Running Test Cases in Parallel	209
Testing Multiple Machines	217
Testing Multiple Applications	223
Troubleshooting Distributed Testing	232
Testing Apache Flex Applications	232
Overview of Apache Flex Support	233
Configuring Security Settings for Your Local Flash Player	233
Configuring Flex Applications to Run in Adobe Flash Player	234
Configuring Flex Applications for Adobe Flash Player Security Restrictions	234
Customizing Apache Flex Scripts	235
Styles in Apache Flex Applications	235
Locator Attributes for Apache Flex Controls	236
Dynamically Invoking Apache Flex Methods	236
Testing Multiple Flex Applications on the Same Web Page	237
Adobe AIR Support	238
Apache Flex Exception Values	238
Overview of the Flex Select Method Using Name or Index	238
Selecting an Item in the FlexDataGrid Control	239
Enabling Your Flex Application for Testing	240
Testing the Component Explorer	250
Testing Flex Custom Controls	254
Client/Server Application Support	263
Client/Server Testing Challenges	263
Verifying Tables in ClientServer Applications	263
Evolving a Testing Strategy	264
Incremental Functional Test Design	265
Network Testing Types	265
How 4Test Handles Script Deadlock	266
Troubleshooting Configuration Test Failures	267
Testing .NET Applications with the Open Agent	267
Windows Forms Applications	267
WPF Applications	271
Microsoft Silverlight Applications	277
Testing Java AWT/Swing Applications with the Open Agent	281
Testing Standard Java Objects and Custom Controls	281
Recording and Playing Back JFC Menus	282
Recording and Playing Back Java AWT Menus	282
Object Recognition for Java AWT/Swing Applications	282
Agent Support for Java AWT/Swing Applications	282
Configuring a Test Application that Uses the Java Network Launching Protocol (JNLP)	283
Custom Attributes	283
Locator Attributes for Java AWT/Swing Controls	284
Dynamically Invoking Java Methods	284
Determining the priorLabel in the Java AWT/Swing Technology Domain	285
Supported Browsers for Testing Java Applets	285
Overview of JavaScript Support	285
Oracle Forms Support	286
Classes in Object-Oriented Programming Languages	287

Configuring Silk Test Classic to Test Java	287
Testing Java Applications and Applets	290
Frequently Asked Questions About Testing Java Applications	296
Testing Java SWT and Eclipse Applications with the Open Agent	297
Suppressing Controls (Open Agent)	298
Custom Attributes	298
Locator Attributes for Java SWT Controls	299
Dynamically Invoking Java Methods	299
Java SWT Classes for the Open Agent	300
Troubleshooting Java SWT and Eclipse Applications	300
Testing Mobile Applications	300
Android	301
iOS	304
Testing an Installed App	311
Recording Mobile Applications	311
Selecting the Mobile Device for Test Replay	312
Connection String for a Mobile Device	312
Interacting with a Mobile Device	315
Releasing a Mobile Device	316
Troubleshooting when Testing Mobile Applications	317
Limitations for Testing Mobile Web Applications	320
Limitations for Testing Native Mobile Applications	321
Clicking on Objects in a Mobile Website	322
Using Existing Mobile Web Tests	323
Testing Rumba Applications	323
Enabling and Disabling Rumba	323
Locator Attributes for Identifying Rumba Controls	324
Testing a Unix Display	324
Rumba Class Reference	324
Testing SAP Applications	324
Locator Attributes for SAP Controls	325
Dynamically Invoking SAP Methods	325
Configuring Automation Security Settings for SAP	326
SAP Class Reference	326
Cross-Browser Testing with the Open Agent	326
Test Objects for xBrowser	328
Overview of Test Frames	328
Testing Web Applications on Different Browsers	329
Changing the Browser Type When Replaying Tests from the UI	330
Setting the Browser when Running Automated Tests	330
Testing Objects in a Web Page with the Open Agent	331
Object Recognition for xBrowser Objects	334
xBrowser Default BaseState	335
Locator Attributes for xBrowser controls	335
Page Synchronization for xBrowser	336
Configuring the Locator Generator for xBrowser	338
Comparing API Playback and Native Playback for xBrowser	339
Setting Recording Options for xBrowser	339
Browser Configuration Settings for xBrowser	340
Connection String for a Remote Desktop Browser	341
Testing with Apple Safari on a Mac	342
Testing with Google Chrome	345
Testing with Mozilla Firefox	348
Testing with Microsoft Edge	351
Manually Creating Tests for Dynamic Popup Menus	353
Finding Hidden Input Fields	353

Cross-Browser Testing: Frequently Asked Questions	353
Testing the Insurance Company Sample Web Application	357
xBrowser Classes	360
Testing Windows API-Based Applications	360
Overview of Windows API-Based Application Support	360
Locator Attributes for Windows API-Based Applications	360
Suppressing Controls (Classic Agent)	361
Suppressing Controls (Open Agent)	361
Configuring Standard Applications	362
Determining the priorLabel in the Win32 Technology Domain	363
Limitations for Testing on Microsoft Windows 10	363
Keyword-Driven Tests	364
Advantages of Keyword-Driven Testing	364
Keywords	365
Creating a Keyword-Driven Test in Silk Test Classic	366
Recording a Keyword-Driven Test in Silk Test Classic	366
Setting the Base State for a Keyword-Driven Test in Silk Test Classic	367
Implementing a Keyword in Silk Test Classic	368
Recording a Keyword in Silk Test Classic	368
Marking a Test Method in a Script as a Keyword	368
Editing a Keyword-Driven Test	369
Managing Keywords in a Test in Silk Central	370
Which Keywords Does Silk Test Classic Recommend?	371
Using Parameters with Keywords	372
Example: Keywords with Parameters	372
Combining Keywords into Keyword Sequences	374
Replaying a Keyword-Driven Test with Specific Variables	374
Integrating Silk Test Classic with Silk Central	375
Implementing Silk Central Keywords in Silk Test Classic	375
Uploading a Keyword Library to Silk Central	376
Uploading a Keyword Library to Silk Central from the Command Line	376
Searching for a Keyword	377
Filtering Keywords	377
Using Advanced Techniques with the Open Agent	378
Starting from the Command Line	378
Starting Silk Test Classic from the Command Line	378
Recording a Test Frame	380
Overview of Object Files	380
Declarations	382
Window Declarations	385
Overview of Identifiers	387
Save the Test Frame	388
Specifying How a Dialog Box is Invoked	388
Improving Object Recognition with Microsoft Accessibility	389
Using Accessibility with the Open Agent	389
Enabling Accessibility for the Open Agent	389
Calling Windows DLLs from 4Test	390
Aliasing a DLL Name	390
Calling a DLL from within a 4Test Script	390
Passing Arguments to DLL Functions	391
Using DLL Support Files Installed with Silk Test Classic	393
Extending the Class Hierarchy	393
Classes	393
Verifying Attributes and Properties	398
Defining Methods and Custom Properties	400
Examples	403

Porting Tests to Other GUIs	405
Handling Differences Among GUIs	405
About GUI Specifiers	409
Supporting GUI-Specific Objects	412
Supporting Custom Controls	414
Why Silk Test Classic Sees Controls as Custom Controls	414
Reasons Why Silk Test Classic Sees the Control as a Custom Control	414
Supporting Graphical Controls	415
Custom Controls (Open Agent)	415
Using Clipboard Methods	420
Filtering Custom Classes	421
Supporting Internationalized Objects	422
Overview of Silk Test Classic Support of Unicode Content	422
Using DB Tester with Unicode Content	423
Issues Displaying Double-Byte Characters	423
Learning More About Internationalization	423
Silk Test Classic File Formats	423
Working with Bi-Directional Languages	426
Configuring Your Environment	426
Troubleshooting Unicode Content	428
Using Autocomplete	430
Overview of AutoComplete	430
Customizing your MemberList	431
Frequently Asked Questions about AutoComplete	432
Turning AutoComplete Options Off	433
Using AppStateList	433
Using DataTypeList	434
Using FunctionTip	434
Using MemberList	434
Overview of the Library Browser	435
Library Browser Source File	435
Adding Information to the Library Browser	436
Add User-Defined Files to the Library Browser with Silk Test Classic	437
Viewing Functions in the Library Browser	437
Viewing Methods for a Class in the Library Browser	437
Examples of Documenting User-Defined Methods	437
Web Classes Not Displayed in Library Browser	438
Library Browser	438
Text Recognition Support	439
Slowing Down Tests	441
Testing Applications in Multiple UI Sessions on a Single Machine	441
Running Tests and Interpreting Results	443
Running Tests	443
Creating a Suite	443
Passing Arguments To a Script	443
Running a Test Case	444
Running a Test Plan	445
Running the Currently Active Script or Suite	445
Running Tests in Parallel	446
Setting Replay Options for the Open Agent	446
Stopping a Running Test Case Before it Completes	447
Setting a Test Case to Use Animation Mode	447
Run Application State Dialog Box	447
Run Testcase Dialog Box	447
Runtime Status Dialog Box	448
Interpreting Results	449

Overview of the Results File	449
Viewing Test Results	450
Errors And the Results File	450
Viewing Differences	452
Testplan Pass/Fail Report and Chart	452
Merging Test Plan Results	452
How Does Silk Test Classic Synchronize Tests?	453
Selecting which Results to Display	454
Select Results to Chart Dialog Box	455
Export Results Dialog Box	455
View Options Dialog Box	455
Results Pass/Fail Report Dialog Box	456
Compare Two Results Dialog Box	456
Analyzing Results with the Silk TrueLog Explorer	456
TrueLog Explorer	457
TrueLog Limitations and Prerequisites	457
Opening the TrueLog Options Dialog Box	458
Setting TrueLog Options	458
Toggle TrueLog at Runtime Using a Script	459
Viewing Results Using the TrueLog Explorer	459
Modifying Your Script to Resolve Window Not Found Exceptions When Using TrueLog	460
Analyzing Bitmaps	460
Overview of the Bitmap Tool	460
When to use the Bitmap Tool	461
Capturing Bitmaps with the Bitmap Tool	461
Comparing Bitmaps	463
Rules for Using Comparison Commands	464
Bitmap Functions	464
Baseline and Result Bitmaps	464
Zooming the Baseline Bitmap, Result Bitmap, and Differences Window	465
Looking at Statistics	465
Exiting from Scan Mode	465
Starting the Bitmap Tool	466
Using Masks	466
Analyzing Bitmaps for Differences	469
Working with Result Files	470
Attaching a Comment to a Result Set	470
Comparing Result Files	470
Customizing results	471
Deleting Results	471
Change the default number of result sets	471
Changing the Colors of Elements In the Results File	472
Fix incorrect values in a script	472
Marking Failed Test Cases	472
Merging Test Plan Results	472
Navigating to errors	473
Viewing an individual summary	473
Storing and Exporting Results	473
Storing results	473
Extracting Results	474
Exporting Results	474
Removing the Unused Space from a Result File	475
Logging Elapsed Time, Thread, and Machine Information	475
Presenting Results	475
Fully customize a chart	475

Generate a Pass/Fail Report on the Active Test Plan Results File	476
Producing a Pass/Fail Chart	476
Displaying a different set of results	477
Debugging Test Scripts	478
Designing and Testing with Debugging in Mind	478
Executing a Script in the Debugger	479
Debugging a Test Script	479
Debugger Menus	480
Stepping Into and Over Functions	480
Working with Scripts During Debugging	480
Exiting the Debugger	480
Breakpoints	481
Setting Breakpoints	481
Viewing Breakpoints	481
Deleting Breakpoints	482
Add Breakpoint Dialog Box	482
Delete Breakpoint Dialog Box	482
Breakpoint Dialog Box	482
Viewing Variables	482
Changing the Value of a Variable	483
Globals Dialog Box	483
Locals Dialog Box	483
Expressions	483
Evaluating Expressions	484
Enabling View Trace Listing	484
Viewing a List of Modules	484
View Module Dialog Box	485
Viewing the Debugging Transcripts	485
Transcript Dialog Box	485
Call Stack Dialog Box	485
Debugging Tips	485
Checking the Precedence of Operators	485
Checking for Code that Never Executes	486
Global and Local Variables with the Same Name	486
Handling Global Variables with Unexpected Values	486
Incorrect Usage of Break Statements	486
Incorrect Values for Loop Variables	486
Infinite loops	486
Typographical Errors	486
Uninitialized Variables	487
Setting Silk Test Classic Options	488
Setting General Options	488
Setting the Editor Font	490
Setting the Editor Colors	490
Runtime Options Dialog Box	491
Compiler Constants Dialog Box	494
Agent Options Dialog Box	494
Timing Tab	495
Verification Tab	496
Close Tab	497
Bitmap Tab	498
Synchronization Tab	499
Advanced Tab	500
Other Tab	501
Compatibility Tab	502
Extensions Dialog Box	503

Extension Details Dialog Box	505
Setting Recording Options for the Open Agent	505
Setting Recording Options for the Classic Agent	507
Setting Replay Options for the Open Agent	508
Defining which Custom Locator Attributes to Use for Recognition	508
Setting Classes to Ignore	509
Custom Controls Dialog Box	509
Property Sets Dialog Box	510
New Property Set Dialog Box	511
Combine Property Sets Dialog Box	511
DOM Extensions Dialog Box	512
Extension Application Dialog Box	513
Extension Options (ActiveX) Dialog Box	514
Extension Options Dialog Box (Java)	514
TrueLog Options - Classic Agent Dialog Box	514
TrueLog Options - Open Agent Dialog Box	517
Troubleshooting the Open Agent	519
Troubleshooting Apache Flex Applications	519
Why Cannot Silk Test Classic Recognize Apache Flex Controls?	519
Troubleshooting Basic Workflow Issues with the Open Agent	519
Error Messages	519
Agent not responding	520
Control is not responding	520
Functionality Not Supported on the Open Agent	520
Unable to Connect to Agent	521
Window is not active	521
Window is not enabled	522
Window is not exposed	522
Window not found	522
Handling Exceptions	523
Default Error Handling	523
Custom Error Handling	523
Trapping the exception number	525
Defining Your Own Exceptions	525
Using do...except Statements to Trap and Handle Exceptions	526
Programmatically Logging an Error	526
Performing More than One Verification in a Test Case	527
Writing an Error-Handling Function	529
Exception Values	530
Troubleshooting Java Applications	534
What Can I Do If the Silk Test Java File Is Not Included in a Plug-In?	534
What Can I Do If Java Controls In an Applet Are Not Recognized?	534
Multiple Machines Testing	534
Setting Up the Recovery System for Multiple Local Applications	534
two_apps.t	535
two_apps.inc	536
Other Problems	541
Adding a Property to the Recorder	541
Cannot Double-Click a Silk Test Classic File and Open Silk Test Classic	542
Cannot Extend AnyWin, Control, or MoveableWin Classes	542
Cannot open results file	543
Common DLL Problems	543
Common Scripting Problems	544
Conflict with Virus Detectors	545
Displaying the Euro Symbol	545
Do I Need Administrator Privileges to Run Silk Test Classic?	546

General Protection Faults	546
Running Global Variables from a Test Plan Versus Running Them from a Script	547
Include File or Script Compiles but Changes are Not Picked Up	547
Library Browser Not Displaying User-Defined Methods	548
Maximum Size of Silk Test Classic Files	548
Recorder Does Not Capture All Actions	549
Relationship between Exceptions Defined in 4test.inc and Messages Sent To the Result File	549
The 4Test Editor Does Not Display Enough Characters	549
Stopping a Test Plan	550
Using a Property Instead of a Data Member	550
Using File Functions to Add Information to the Beginning of a File	550
Why Does the Str Function Not Round Correctly?	551
Troubleshooting Projects	551
Files Not Found When Opening Project	551
Silk Test Classic Cannot Load My Project File	551
Silk Test Classic Cannot Save Files to My Project	552
Silk Test Classic Does Not Run	552
My Files No Longer Display In the Recent Files List	552
Cannot Find Items In Classic 4Test	553
Editing the Project Files	553
Recognition Issues	553
How Can the Application Developers Make Applications Ready for Automated Testing?	553
Tips	554
Example Test Cases for the Find Dialog Box	554
When to use the Bitmap Tool	555
Troubleshooting Web Applications	555
What Can I Do If the Page I Have Selected Is Empty?	555
Why Do I Get an Error Message When I Set the Accessibility Extension?	555
Using the Runtime Version of Silk Test Classic	556
Installing the Runtime Version	556
Starting the Runtime Version	556
Comparing Silk Test Classic and Silk Test Classic Runtime Menus and Commands	556
Working with Files	567
Creating a New File	567
Searching for a String in a File	567
Replacing a String in a File	568
4Test Editor	568
Setting Up a Printer	569
Printing the Contents of the Active Window	569
Confirm Test Description Identifier Dialog Box	570
Glossary	571
4Test Classes	571
4Test-Compatible Information or Methods	571
Abstract Windowing Toolkit	571
accented character	571
agent	571
applet	572
application state	572
attributes	572
Band (.NET)	572
base state	572
bidirectional text	572

Bytecode	572
call stack	573
child object	573
class	573
class library	573
class mapping	573
Classic 4Test	573
client area	573
custom object	573
data-driven test case	574
data member	574
declarations	574
DefaultBaseState	574
diacritic	574
Difference Viewer	574
double-byte character set (DBCS)	574
dynamic instantiation	574
dynamic link library (DLL)	575
enabling	575
exception	575
frame file	575
fully qualified object name	575
group description	575
handles	576
hierarchy of GUI objects	576
host machine	576
hotkey	576
Hungarian notation	580
identifier	580
include file	581
internationalization or globalization	581
Java Database Connectivity (JDBC)	581
Java Development Kit (JDK)	581
Java Foundation Classes (JFC)	581
Java Runtime Environment (JRE)	581
Java Virtual Machine (JVM)	581
JavaBeans	581
Latin script	582
layout	582
levels of localization	582
load testing	582
localization	582
localize an application	582
locator	582
logical hierarchy	583
manual test	583
mark	583
master plan	583
message box	583
method	583
minus (-) sign	583
modal	583
modeless	584
Multibyte Character Set (MBCS)	584
Multiple Application Domains (.NET)	584
negative testing	584

nested declarations	584
No-Touch (.NET)	584
object	584
outline	584
Overloaded method	585
parent object	585
performance testing	585
physical hierarchy (.NET)	585
plus (+) sign	585
polymorphism	585
project	585
properties	586
query	586
recovery system	586
regression testing	586
results file	586
script	586
script file	586
side-by-side (.NET)	586
Simplified Chinese	587
Single-Byte Character Set (SBCS)	587
smoke test	587
Standard Widget Toolkit (SWT)	587
statement	587
status line	587
stress testing	587
subplan	588
suite	588
Swing	588
symbols	588
tag	588
target machine	588
template	589
test description	589
test frame file	589
test case	589
test plan	589
TotalMemory parameter	589
Traditional Chinese	590
variable	590
verification statement	590
Visual 4Test	590
window declarations	590
window part	590
XPath	590



Licensing Information

Unless you are using a trial version, Silk Test requires a license.



Note: A Silk Test license is bound to a specific version of Silk Test. For example, Silk Test 17.5 requires a Silk Test 17.5 license.

The licensing model is based on the client that you are using and the applications that you want to be able to test. The available licensing modes support the following application types:

Licensing Mode	Application Type
Mobile Native	<ul style="list-style-type: none"> • Mobile web applications. <ul style="list-style-type: none"> • Android • iOS • Native mobile applications. <ul style="list-style-type: none"> • Android • iOS
Full	<ul style="list-style-type: none"> • Web applications, including the following: <ul style="list-style-type: none"> • Apache Flex • Java-Applets • Mobile web applications. <ul style="list-style-type: none"> • Android • iOS • Apache Flex • Java AWT/Swing, including Oracle Forms • Java SWT and Eclipse RCP • .NET, including Windows Forms and Windows Presentation Foundation (WPF) • Rumba • Windows API-Based <p> Note: To upgrade your license to a Full license, visit www.borland.com.</p>
Premium	<p>All application types that are supported with a <i>Full</i> license, plus SAP applications.</p> <p> Note: To upgrade your license to a Premium license, visit www.borland.com.</p>
Mobile Native Add-On	<p>In addition to the technologies supported with a Full or Premium license, the mobile native add-on license offers support for testing native mobile applications on Android and iOS.</p>

Getting Started

Silk Test Classic is the traditional Silk Test client. With Silk Test Classic you can develop tests using the 4Test language, an object-oriented fourth-generation language (4GL), which is designed specifically for QA professionals. Silk Test Classic guides you through the entire process of creating test cases, running the tests, and interpreting the results of your test runs.

Silk Test Classic supports the testing of a broad set of application technologies.

This section provides information to get you up and running with Silk Test Classic.



Note: If you have opted not to display the start screen when you start Silk Test Classic, you can check for available updates by clicking **Help > Check for Product Update**.

Automation Under Special Conditions (Missing Peripherals)

Basic product orientation

Silk Test Classic is a GUI testing product that tries to act like a human user in order to achieve meaningful test results under automation conditions. A test performed by Silk Test Classic should be as valuable as a test performed by a human user while executing much faster. This means that Silk Test Classic requires a testing environment that is as similar as possible to the testing environment that a human user would require in order to perform the same test.

Physical peripherals

Manually testing the UI of a real application requires physical input and output devices like a keyboard, a mouse, and a display. Silk Test Classic does not necessarily require physical input devices during test replay. What Silk Test Classic requires is the ability of the operating system to perform keystrokes and mouse clicks. The Silk Test Classic replay usually works as expected without any input devices connected. However, some device drivers might block the Silk Test Classic replay mechanisms if the physical input device is not available.

The same applies to physical output devices. A physical display does not necessarily need to be connected, but a working video device driver must be installed and the operating system must be in a condition to render things to the screen. For example, rendering is not possible in screen saver mode or if a session is locked. If rendering is not possible, low-level replay will not work and high-level replay might also not work as expected, depend on the technology that is used in the application under test (AUT).

Virtual machines

Silk Test Classic does not directly support virtualization vendors, but can operate with any type of virtualization solution as long as the virtual guest machine behaves like a physical machine. Standard peripherals are usually provided as virtual devices, regardless of which physical devices are used with the machine that runs the virtual machine.

Cloud instances

From an automation point of view, a cloud instance is not different to a virtual machine. However, a cloud instance might run some special video rendering optimization, which might lead to situations where screen rendering is temporarily turned off to save hardware resources. This might happen when the cloud instance detects that no client is actively viewing the display. In such a case, you could open a VNC window as a workaround.

Special cases

Application launched without any window (headless)

Such an application cannot be tested with Silk Test Classic. Silk Test Classic needs to hook to a target application process in order to interact with it. Hooking is not possible for processes that do not have a visible window. In such a case you can only run system commands.

Remote desktops, terminal services, and remote applications (all vendors)

If Silk Test Classic resides and operates within a remote desktop session, it will fully operate as expected.



Note: You require a full user session and the remote viewing window needs to be maximized. If the remote viewing window is not displayed for some reason, for example network issues, Silk Test Classic will continue to replay but might produce unexpected results, depending on what remote viewing technology is used. For example, a lost remote desktop session will negatively impact video rendering, whereas other remote viewing solutions might show no impact at all once the viewing window was lost.

If Silk Test Classic is used to interact with the remote desktop, remote view, or remote app window, only low-level techniques can be used, because Silk Test Classic sees only a screenshot of the remote machine. For some remote viewing solutions even low-level operations may not be possible because of security restrictions. For example, it might not be possible to send keystrokes to a remote application window.

Known automation obstacles

Silk Test Classic requires an interactively-logged-on full-user session. Disable anything that could lock the session, for example screen savers, hibernation, or sleep mode. If this is not possible because of organizational policies you could workaround such issues by adding *keep alive* actions, for example moving the mouse, in regular intervals or at the end of each test case.



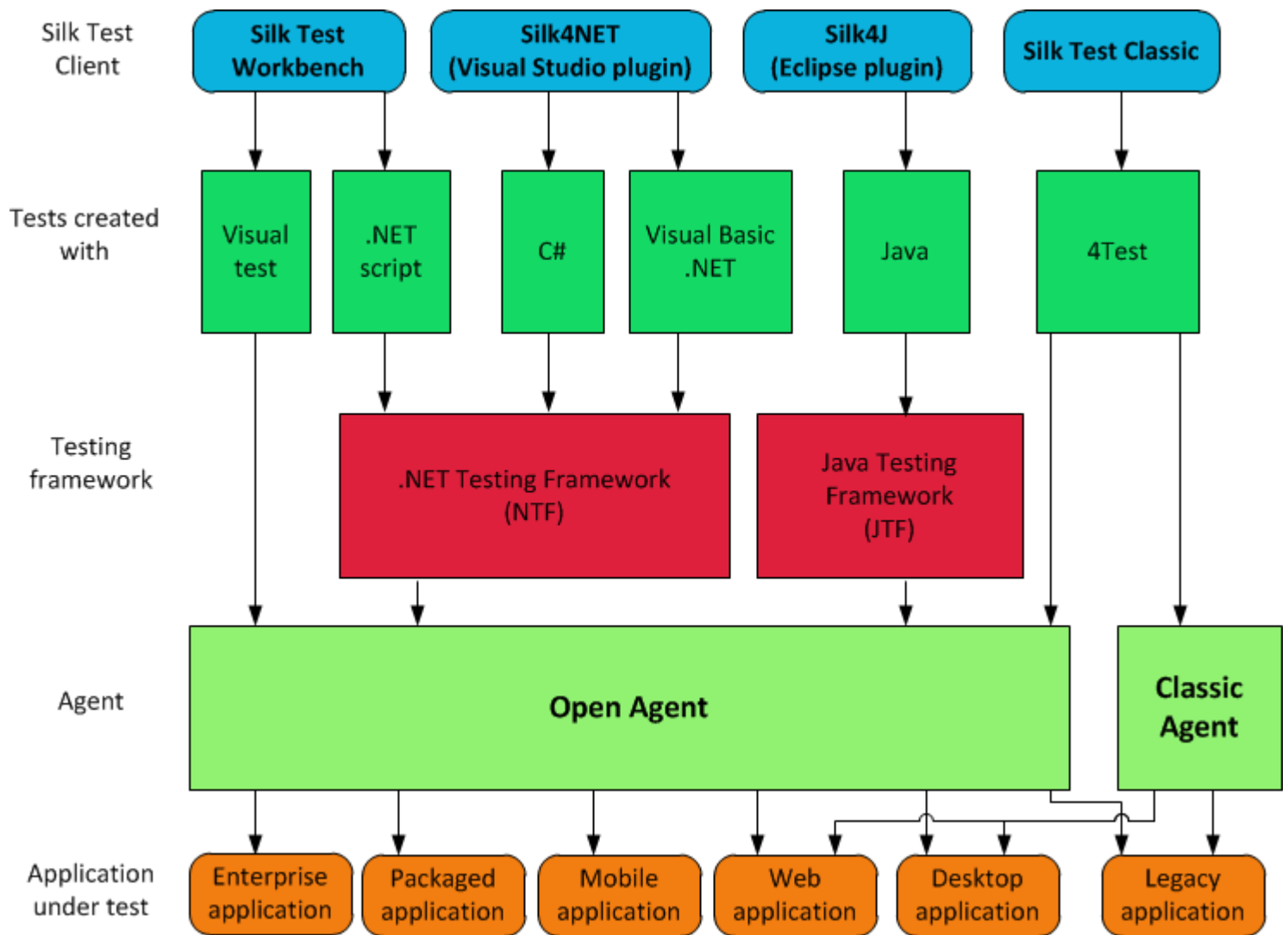
Note: Depending on the configuration of the actual testing environment and the technologies that are used for the AUT, the virtualization, and the terminal services, you may face additional challenges and limitations during the test automation process.

Silk Test Product Suite

Silk Test is an automated testing tool for fast and reliable functional and regression testing. Silk Test helps development teams, quality teams, and business analysts to deliver software faster, and with high quality. With Silk Test you can record and replay tests across multiple platforms and devices to ensure that your applications work exactly as intended.

The Silk Test product suite includes the following components:

- Silk Test Workbench – Silk Test Workbench is the quality testing environment that offers .NET scripting for power users and easy to use visual tests to make testing more accessible to a broader audience.
- Silk4NET – The Silk4NET Visual Studio plug-in enables you to create Visual Basic or C# test scripts directly in Visual Studio.
- Silk4J – The Silk4J Eclipse plug-in enables you to create Java-based test scripts directly in your Eclipse environment.
- Silk Test Classic – Silk Test Classic is the Silk Test client that enables you to create scripts based on 4Test.
- Silk Test Agents – The Silk Test agent is the software process that translates the commands in your tests into GUI-specific commands. In other words, the agent drives and monitors the application you are testing. One agent can run locally on the host machine. In a networked environment, any number of agents can run on remote machines.



The sizes of the individual boxes in the image above differ for visualization purposes and do not reflect the included functionality.

The product suite that you install determines which components are available. To install all components, choose the complete install option. To install all components with the exception of Silk Test Classic, choose the standard install option.

Contacting Micro Focus

Micro Focus is committed to providing world-class technical support and consulting services. Micro Focus provides worldwide support, delivering timely, reliable service to ensure every customer's business success.

All customers who are under a maintenance and support contract, as well as prospective customers who are evaluating products, are eligible for customer support. Our highly trained staff respond to your requests as quickly and professionally as possible.

Visit <http://supportline.microfocus.com/assistedservices.asp> to communicate directly with Micro Focus SupportLine to resolve your issues, or email supportline@microfocus.com.

Visit Micro Focus SupportLine at <http://supportline.microfocus.com> for up-to-date support news and access to other support information. First time users may be required to register to the site.

Information Needed by Micro Focus SupportLine

When contacting Micro Focus SupportLine, please include the following information if possible. The more information you can give, the better Micro Focus SupportLine can help you.

- The name and version number of all products that you think might be causing an issue.
- Your computer make and model.
- System information such as operating system name and version, processors, and memory details.
- Any detailed description of the issue, including steps to reproduce the issue.
- Exact wording of any error messages involved.
- Your serial number.

To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

What's New in Silk Test Classic

Silk Test Classic supports the following new features for the Open Agent:

Mobile Device Live View

Do you need to see what is currently happening on a mobile device, even if the device is in another room, another building, or even another country? While you record or replay on a mobile device or emulator, Silk Test Classic mirrors the display of the mobile device on the desktop, enabling you to see every action that is performed on the device in real time.

Testing Hybrid Mobile Applications

Hybrid mobile applications (apps) use web technologies wrapped in native mobile apps to ease building cross-device compatible apps. Silk Test Classic now supports testing such apps on Android and iOS, by exposing both the full Silk Test Classic browser testing API as well as the full native mobile API.

Enhanced Cross-Browser Testing

Silk Test 17.5 features the following cross browser enhancements:

- To enable you to incorporate the browser-specific features of Google Chrome in your web application testing, Silk Test 17.5 introduces action and locator recording support for Google Chrome.
- With Silk Test 17.5, the replay performance of cross-browser scripts in Mozilla Firefox, Google Chrome, and Microsoft Edge is significantly improved.

Using Devices and Browsers that are Managed by Silk Central

You can now execute your mobile and web tests on mobile devices and browsers that are managed by Silk Central. Just add Silk Central as a remote location and start testing on all managed devices and browsers.

Usability Enhancements

This section lists usability enhancements that have been made in Silk Test 17.5.

Enhanced action recording for mobile and web applications

Silk Test enables you to record the default action for an object without having to choose. This makes action recording faster and easier. While recording actions against a mobile application or a web application in Microsoft Edge, Mozilla Firefox, Google Chrome, or Apple Safari, you no longer have to select every action in the **Choose Action** dialog. However, the **Choose Action** dialog continues to open for objects that do not offer a default action, for example text fields, and you can still open the **Choose Action** dialog by right-clicking on any object.

Disabling remote locations

To decrease the amount of browsers and devices displayed in the **Select Application** dialog, you can now enable or disable the remote locations in the **Edit Remote Locations** dialog. When disabling a location, the installed browsers and connected devices are no longer displayed in the **Select Application** dialog.

Multiple application domain support for WPF

Multi application domain enables you to design and implement secure .NET WPF applications. Silk Test Classic now supports testing WPF applications that use multiple application domains.

Ignoring specific WPF classes

To generate compact and easy to read locators for WPF controls, you can now specify any WPF classes that Silk Test Classic should ignore as transparent classes.

Clicking on a specified location on an image

When you perform an image click, Silk Test by default clicks into the center of the image asset. This might not always be the best choice. With Silk Test 17.5, you can specify the exact location on which clicks should be performed in the image asset UI.

Customizing TrueLog file names

You can now add placeholders to the TrueLog file name. These placeholders are replaced with the appropriate data at execution time.

API Enhancements

This section lists API enhancements that have been made in Silk Test 17.5.

Setting the geographic location of a mobile device

You can now use the `SetLocation` method of the `MobileDevice` class to set the geographic location of a mobile device.

Testing cross-domain IFrames with Google Chrome

To test cross-domain IFrames with Google Chrome 49 or later, you do no longer have to set the options `OPT_XBROWSER_GOOGLECHROME_USER_DATA_DIR` and `OPT_XBROWSER_GOOGLECHROME_SECURE_ORIGINS`. Silk Test now supports testing such IFrames with Google Chrome out-of-the-box.



Note: As these options have been removed from Silk Test, remove any code for setting these options from any existing scripts for cross-domain IFrame testing with Google Chrome.

Technology Updates

This section lists the significant technology updates for Silk Test 17.5.

Microsoft Windows 10 Anniversary Update

With Silk Test 17.5, you can leverage your test set by simply executing all your existing functional, regression, and localization tests on Microsoft Windows 10 Anniversary Update.

Additionally, Silk Test 17.5 offers support for Microsoft Edge 38.14393, the new browser version for Microsoft Windows 10 Anniversary Update.

New Mozilla Firefox Versions

In addition to the versions of Mozilla Firefox, which have been tested with the previous version of Silk Test, Silk Test has been tested for recording and replay with the following new versions of Mozilla Firefox:

- Mozilla Firefox 47
- Mozilla Firefox 48
- Mozilla Firefox 49



Note: This list includes the new versions of Mozilla Firefox that have been tested with Silk Test 17.5 until the release date of Silk Test 17.5. Silk Test 17.5 should be able to support newer versions of Mozilla Firefox, even if these versions have been released after the release date of Silk Test 17.5.

Google Chrome Support

In addition to the versions of Google Chrome, which have been tested with the previous version of Silk Test, Silk Test playback has now been tested with the following versions of Google Chrome:

- Google Chrome 51
- Google Chrome 52
- Google Chrome 53
- Google Chrome 54



Note: This list includes the versions of Google Chrome that have been tested with Silk Test 17.5 until the release date of Silk Test 17.5. Silk Test 17.5 should be able to support newer versions of Google Chrome, even if these versions have been released after the release date of Silk Test 17.5.

Android Support

In addition to the Android versions, which have been tested with the previous version of Silk Test, Silk Test now includes support for mobile web applications and native mobile applications for the following operating systems:

- Android 7.0

Java SWT Support

Silk Test now supports Java SWT 4.6 standalone and Rich Client Platform (RCP) applications.

Open Agent

The Silk Test agent is the software process that translates the commands in your test scripts into GUI-specific commands. In other words, the agent drives and monitors the application you are testing. One agent can run locally on the host machine. In a networked environment, any number of agents can run on remote machines.

Silk Test Classic provides two types of agents, the Open Agent and the Classic Agent. The agent that you assign to your project or script depends on the type of application that you are testing.

When you create a new project, Silk Test Classic automatically uses the agent that supports the type of application that you are testing. For instance, if you create an Apache Flex or Windows API-based client/server project, Silk Test Classic uses the Open Agent. When you open a project or script that was developed with the Classic Agent, Silk Test Classic automatically uses the Classic Agent. For information about the supported technology domains for each agent, refer to *Testing in Your Environment*.

The Open Agent supports dynamic object recognition to record and replay test cases that use XPath queries to find and identify objects. With the Open Agent, one Agent can run locally on the host machine. In a networked environment, any number of Agents can replay tests on remote machines. However, you can record only on a local machine.


For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

How Silk Test Classic Assigns an Agent to a Window Declaration

When you record a test with the Open Agent set as the default agent, Silk Test Classic includes a locator to identify the top-most window of the test application. For instance, this window declaration for a Notepad application that uses the Open Agent includes the following locator:

```
window MainWin UntitledNotepad
locator "/MainWin[@caption='Untitled - Notepad']"
```

Silk Test Classic determines which agent to use by detecting whether a locator or `Find` or `FindAll` command is used. When Silk Test Classic detects a locator on the top-most window or detects a `Find` or `FindAll` command, the Open Agent is automatically used. If no locator or `Find` or `FindAll` command is present, Silk Test Classic uses the Classic Agent.

 **Note:** Any window declaration is only valid for either the Open Agent or the Classic Agent. There is no way to use the same window declaration with both agents. The only exception to this rule are `SYS` functions and `DLL` calls, which are implemented for both agents.

Agent Options

The following table lists the `AgentClass` options that can be manipulated with the `getOption` method and `setOption` method. Only options that can be manipulated by the user are listed here; other options are for internal use only.


Agent Option	Agent Supported	Description
OPT_AGENT_CLICKS_ONLY	Classic Agent	BOOLEAN

Agent Option	Agent Supported	Description
		<p>FALSE to use the API-based clicks; TRUE to use agent-based clicks. The default is FALSE. This option applies to clicks on specific HTML options only. For additional information, see <i>API Click Versus Agent Click</i>.</p> <p>This option can be set through the Compatibility tab on the Agent Options dialog box, <code>Agent.SetOption</code>, or <code>BindAgentOption()</code>, and may be retrieved through <code>Agent.GetOption()</code>.</p>
OPT_ALTERNATE_RECORD_BREAK	Classic Agent Open Agent	BOOLEAN TRUE pauses recording when Ctrl+Shift is pressed. Otherwise, Ctrl+Alt is used. By default, this is FALSE.
OPT_APPREADY_RETRY	Classic Agent Open Agent	NUMBER The number of seconds that the agent waits between attempts to verify that an application is ready. The agent continues trying to test the application for readiness if it is not ready until the time specified with <code>OPT_APPREADY_TIMEOUT</code> is reached.
OPT_APPREADY_TIMEOUT	Classic Agent Open Agent	NUMBER The number of seconds that the agent waits for an application to become ready. If the application is not ready within the specified timeout, Silk Test Classic raises an exception. To require the agent to check the ready state of an application, set <code>OPT_VERIFY_APPREADY</code> . This option applies only if the application or extension knows how to communicate to the agent that it is ready. To find out whether the extension has this capability, see the documentation that comes with the extension.
OPT_BITMAP_MATCH_COUNT	Classic Agent Open Agent	INTEGER The number of consecutive snapshots that must be the same for the bitmap to be considered stable. Snapshots are taken up to the number of

Agent Option	Agent Supported	Description
		<p>seconds specified by OPT_BITMAP_MATCH_TIMEOUT, with a pause specified by OPT_BITMAP_MATCH_INTERVAL occurring between each snapshot.</p> <p>Related methods:</p> <ul style="list-style-type: none"> • CaptureBitmap • GetBitmapCRC • SYS_CompareBitmap • VerifyBitmap • WaitBitmap
OPT_BITMAP_MATCH_INTERVAL	<p>Classic Agent</p> <p>Open Agent</p>	<p>INTEGER</p> <p>The time interval between snapshots to use for ensuring the stability of the bitmap image. The snapshots are taken up to the time specified by OPT_BITMAP_MATCH_TIMEOUT.</p> <p>Related methods:</p> <ul style="list-style-type: none"> • CaptureBitmap • GetBitmapCRC • SYS_CompareBitmap • VerifyBitmap • WaitBitmap
OPT_BITMAP_MATCH_TIMEOUT	<p>Classic Agent</p> <p>Open Agent</p>	<p>NUMBER</p> <p>The total time allowed for a bitmap image to become stable.</p> <p>During the time period, Silk Test Classic takes multiple snapshots of the image, waiting the number of seconds specified with OPT_BITMAP_MATCH_TIMEOUT between snapshots. If the value returned by OPT_BITMAP_MATCH_TIMEOUT is reached before the number of bitmaps specified by OPT_BITMAP_MATCH_COUNT match, Silk Test Classic stops taking snapshots and raises the exception E_BITMAP_NOT_STABLE.</p> <p>Related methods:</p> <ul style="list-style-type: none"> • CaptureBitmap • GetBitmapCRC • VerifyBitmap

Agent Option	Agent Supported	Description
OPT_BITMAP_PIXEL_TOLERANCE	Classic Agent Open Agent	<ul style="list-style-type: none"> WaitBitmap <p>INTEGER</p> <p>The number of pixels of difference below which two bitmaps are considered to match. If the number of pixels that are different is smaller than the number specified with this option, the bitmaps are considered identical. The maximum tolerance is 32767 pixels.</p> <p>Related methods:</p> <ul style="list-style-type: none"> SYS_CompareBitmap VerifyBitmap WaitBitmap
OPT_CLASS_MAP	Classic Agent Open Agent	<p>LIST OF STRING</p> <p>The class mapping table for custom objects, with each entry in the list in the form <code>custom_class = standard_class</code>.</p>
OPT_CLOSE_CONFIRM_BUTTONS	Classic Agent Open Agent	<p>LIST OF STRING</p> <p>The list of buttons used to close confirmation dialog boxes, which are dialog boxes that display when closing windows with the methods <code>Close</code>, <code>CloseWindows</code>, and <code>Exit</code>.</p>
OPT_CLOSE_DIALOG_KEYS	Classic Agent Open Agent	<p>LIST OF STRING</p> <p>The keystroke sequence used to close dialog boxes with the methods <code>Close</code>, <code>CloseWindows</code>, and <code>Exit</code>.</p>
OPT_CLOSE_MENU_NAME	Classic Agent	<p>STRING</p> <p>A list of strings representing the list of menu items on the system menu used to close windows with the methods <code>Close</code>, <code>CloseWindows</code>, and <code>Exit</code>.</p> <p>Default is <code>Close</code>.</p>
OPT_CLOSE_WINDOW_BUTTONS	Classic Agent Open Agent	<p>LIST OF STRING</p> <p>The list of buttons used to close windows with the methods <code>Close</code>, <code>CloseWindows</code>, and <code>Exit</code>.</p>

Agent Option	Agent Supported	Description
OPT_CLOSE_WINDOW_MENUS	Classic Agent Open Agent	LIST OF STRING The list of menu items used to close windows with the methods <code>Close</code> , <code>CloseWindows</code> , and <code>Exit</code> .
OPT_CLOSE_WINDOW_TIMEOUT	Classic Agent Open Agent	NUMBER The number of seconds that Silk Test Classic waits before it tries a different close strategy for the <code>Close</code> method when the respective window does not close. Close strategies include Alt+F4 or sending the keys specified by <code>OPT_CLOSE_DIALOG_KEYS</code> . By default, this is 2.
OPT_COMPATIBLE_TAGS	Classic Agent	BOOLEAN TRUE to generate and operate on tags compatible with releases earlier than Release 2; FALSE to use the current algorithm. The current algorithm affects tags that use index numbers and some tags that use captions. In general, the current tags are more portable, while the earlier algorithm generates more platform-dependent tags.
OPT_COMPATIBILITY	Open Agent	STRING Enables you to use the behavior of the specified Silk Test Classic version for specific features, when the behavior of these features has changed in a later version. Example strings: <ul style="list-style-type: none"> • 12 • 11.1 • 13.0.1 By default, this option is not set.
OPT_COMPRESS_WHITESPACE	Classic Agent	BOOLEAN TRUE to replace all multiple consecutive white spaces with a single space for comparison of tags. FALSE (the default) to avoid replacing blank characters in this manner. This is intended to provide a way to match tags where the only difference is the number of white spaces between words.

Agent Option	Agent Supported	Description
		<p>If at all possible, use "wildcard " instead of this option.</p> <p>This option can increase test time because of the increased time it takes for compressing of white spaces in both source and target tags. If Silk Test Classic processes an object that has many children, this option may result in increased testing times.</p> <p>The tag comparison is performed in two parts. The first part is a simple comparison; if there is a match, no further action is required. The second part is to compress consecutive white spaces and retest for a match.</p> <p>Due to the possible increase in test time, the most efficient way to use this option is to enable and disable the option as required on sections of the testing that is affected by white space. Do not enable this option to cover your entire test.</p> <p>Tabs in menu items are processed before the actual tags are compared. Do not modify the window declarations of frame files by adding tabs to any of the tags.</p>
OPT_DROPDOWN_PICK_BEFORE_GET	Classic Agent	<p>BOOLEAN</p> <p>TRUE to drop down the combo box before trying to access the content of the combo box. This is usually not needed, but some combo boxes only get populated after they are dropped down. If you are having problems getting the contents of a combo box, set this option to TRUE.</p> <p>Default is FALSE.</p>
OPT_ENABLE_ACCESSIBILITY	<p>Classic Agent</p> <p>Open Agent</p>	<p>BOOLEAN</p> <p>TRUE to enable Accessibility when you are testing a Win32 application and Silk Test Classic cannot recognize objects. Accessibility is designed to enhance object recognition at the class level. FALSE to disable Accessibility.</p> <p> Note: For Mozilla Firefox and Google Chrome, Accessibility is always activated and cannot be deactivated.</p>


Agent Option	Agent Supported	Description
OPT_ENSURE_ACTIVE_WINDOW	Open Agent	<p>Default is FALSE.</p> <p>BOOLEAN</p> <p>TRUE ensures that the main window of the call is active before a call is executed. By default, this is FALSE.</p>
OPT_EXTENSIONS	Classic Agent	<p>LIST OF STRING</p> <p>The list of loaded extensions. Each extension is identified by the name of the .dll or .vxx file associated with the extension.</p> <p>Unlike the other options, OPT_EXTENSIONS is read-only and works only with <code>GetOption()</code>.</p>
OPT_GET_MULTITEXT_KEEP_EMPTY_LINES	Classic Agent	<p>BOOLEAN</p> <p>TRUE returns an empty list if no text is selected. FALSE removes any blank lines within the selected text.</p> <p>By default, this is TRUE.</p>
OPT_ITEM_RECORD	Open Agent	<p>BOOLEAN</p> <p>For SWT applications, TRUE records methods that invoke tab items directly rather than recording the tab folder hierarchy. For example, you might record <code>SWTControls.SWTTabControl1.TabFolder.Select()</code>. If this option is set to FALSE, SWT tab folder actions are recorded. For example, you might record <code>SWTControls.SWTTabControl1.Select("TabFolder")</code>.</p> <p>By default, this is TRUE.</p>
OPT_KEYBOARD_DELAY	Classic Agent Open Agent	<p>NUMBER</p> <p>Default is 0.02 seconds; you can select a number in increments of .001 from .001 to up to 1000 seconds.</p> <p>Be aware that the optimal number can vary, depending on the application that you are testing. For example, if you are testing a Web application, a setting of .001 radically slows down the browser. However, setting this to 0 (zero) may cause basic application testing to fail.</p>
OPT_KEYBOARD_LAYOUT	Classic Agent	<p>STRING</p>


Agent Option	Agent Supported	Description
		<p>Provides support for international keyboard layouts in the Windows environment. Specify an operating-system specific name for the keyboard layout. Refer to the Microsoft Windows documentation to determine what string your operating system expects. Alternatively, use the <code>GetOption</code> method to help you determine the current keyboard layout, as in the following example:</p> <pre>Print (Agent.GetOption (OPT_KEYBOARD_LAYOUT))</pre>
OPT_KILL_HANGING_APPS	<p>Classic Agent Open Agent</p>	<p>BOOLEAN</p> <p>Specifies whether to shutdown the application if communication between the Agent and the application fails or times out. Set this option to TRUE when testing applications that cannot run multiple instances. By default, this is FALSE.</p>
OPT_LOCATOR_ATTRIBUTES_CASE_SENSITIVE	Open Agent	<p>BOOLEAN</p> <p>Set to Yes to add case-sensitivity to locator attribute names, or to No to match the locator names case insensitive.</p>
OPT_MATCH_ITEM_CASE	<p>Classic Agent Open Agent</p>	<p>BOOLEAN</p> <p>Set this option to TRUE to have Silk Test Classic consider case when matching items in combo boxes, list boxes, radio lists, and popup lists, or set this option to FALSE to ignore case differences during execution of a <code>Select</code> method. This option has no effect on a <code>Verify</code> function or a <code>VerifyContents</code> method.</p>
OPT_MENU_INVOKE_POPUP	Classic Agent	<p>STRING</p> <p>The command, keystrokes or mouse buttons, used to display pop-up menus, which are menus that popup over a particular object. To use mouse buttons, specify <code><button1></code>, <code><button2></code>, or <code><button3></code> in the command sequence.</p>
OPT_MENU_PICK_BEFORE_GET	Classic Agent	<p>BOOLEAN</p> <p>TRUE to pick the menu before checking whether an item on it exists,</p>

Agent Option	Agent Supported	Description
		is enabled, or is checked, or FALSE to not pick the menu before checking. When TRUE, you may see menus pop up on the screen even though your script does not explicitly call the Pick method. Default is FALSE.
OPT_MOUSE_DELAY	Classic Agent Open Agent	NUMBER The delay used before each mouse event in a script. The delay affects moving the mouse, pressing buttons, and releasing buttons. By default, this is 0.02.
OPT_MULTIPLE_TAGS	Classic Agent Open Agent	BOOLEAN TRUE to use multiple tags when recording and playing back. FALSE to use one tag only, as done in previous releases. This option cannot be set through the Agent Options dialog box. Its default is TRUE and is only set by the INI file, option file, and through Agent.SetOption. This option overrides the Record multiple tags check box that displays in both the Recorder Options dialog box and the Record Window Declaration Options dialog box. If the Record multiple tags check box is grayed out and you want to change it, check this setting.
OPT_NO_ICONIC_MESSAGE_BOXES	Classic Agent	BOOLEAN TRUE to not have minimized windows automatically recognized as message boxes. Default is FALSE.
OPT_PAUSE_TRUELOG	Classic Agent	BOOLEAN TRUE to disable TrueLog at runtime for a specific portion of a script, or FALSE to enable TrueLog. This option has no effect if Truelog is not enabled. Default is FALSE.
OPT_PLAY_MODE	Classic Agent	STRING

Agent Option	Agent Supported	Description
		Used to specify playback mechanism. For additional information for Windows applications, see <i>Playing Back Mouse Actions</i> .
OPT_POST_REPLAY_DELAY	Classic Agent Open Agent	NUMBER The time in seconds to wait after invoking a function or writing properties. Increase this delay if you experience replay problems due to the application taking too long to process mouse and keyboard input. By default, this is 0.00.
OPT_RADIO_LIST	Classic Agent	BOOLEAN TRUE to view option buttons as a group; FALSE to use the pre-Release 2 method of viewing option buttons as individual objects.
OPT_RECORD_LISTVIEW_SELECT_BY_TYP EKEYS	Open Agent	BOOLEAN TRUE records methods with typekeys statements rather than with keyboard input for certain selected values. By default, this is FALSE.
OPT_RECORD_MOUSE_CLICK_RADIUS	Open Agent	INTEGER The number of pixels that defines the radius in which a mouse down and mouse up event must occur in order for the Open Agent to recognize it as a click. If the mouse down and mouse up event radius is greater than the defined value, a <code>PressMouse</code> and <code>ReleaseMouse</code> event are scripted. By default, this is set to 5 pixels.
OPT_RECORD_MOUSEMOVES	Classic Agent Open Agent	BOOLEAN TRUE records mouse moves for Web pages, Win32 applications, and Windows Forms applications that use mouse move events. You cannot record mouse moves for child domains of the xBrowser technology domain, for example Apache Flex and Swing. By default, this is FALSE.
OPT_RECORD_SCROLLBAR_ABSOLUT	Open Agent	BOOLEAN TRUE records scroll events with absolute values instead of relative to the previous scroll position. By default, this is FALSE.

Agent Option	Agent Supported	Description
OPT_REL1_CLASS_LIBRARY	Classic Agent	<p>BOOLEAN</p> <p>TRUE to use pre-Release 2 versions of <code>GetChildren</code>, <code>GetClass</code>, and <code>GetParent</code>, or FALSE to use current versions.</p>
OPT_REMOVE_FOCUS_ON_CAPTURE_TEXT	Open Agent	<p>BOOLEAN</p> <p>TRUE to remove the focus from a window before text is captured. By default, this is FALSE.</p>
OPT_REPLAY_HIGHLIGHT_TIME	Open Agent	<p>NUMBER</p> <p>The number of seconds before each invoke command that the object is highlighted.</p> <p>By default, this is 0, which means that objects are not highlighted by default.</p>
OPT_REPLAY_MODE	<p>Classic Agent</p> <p>Open Agent</p>	<p>NUMBER</p> <p>The replay mode defines how replays on a control are executed: They can be executed with mouse and keyboard (low level) or using the API (high level). Each control defines which replay mode is the default mode for the control. When the default replay mode is enabled, most controls use a low level replay. The default mode for each control is the mode that works most reliably. If a replay fails, the user can change the replay mode and try again. Each control that supports that mode will execute the replay in the specified mode. If a control does not support the mode, it executes the default mode. For example, if <code>PushButton</code> supports low level replay but uses high level replay by default, it will use low level replay only if the option specifies it. Otherwise, it will use the high level implementation.</p> <p>Possible values include 0, 1, and 2. 0 is default, 1 is high level, 2 is low level. By default, this is 0.</p>
OPT_REQUIRE_ACTIVE	Classic Agent	<p>BOOLEAN</p> <p>Setting this option to FALSE allows <code>4Test</code> statements to be attempted against inactive windows.</p>

Agent Option	Agent Supported	Description
OPT_RESIZE_APPLICATION_BEFORE_RECORDING	Open Agent	<p>Default is TRUE, except when running script statements that were recorded and are in a recording statement.</p> <p>BOOLEAN</p> <p>Define whether to resize the application under test (AUT) when a recording session starts, to display the recording sidebar next to the AUT. If this option is disabled, the AUT and the recording sidebar might overlap. This option is TRUE by default.</p>
OPT_SCROLL_INTO_VIEW	Classic Agent	<p>BOOLEAN</p> <p>TRUE to scroll a control into view before recording events against it or capturing its bitmap. This option applies only when OPT_SHOW_OUT_OF_VIEW is set to TRUE. This option is useful for testing Web applications in which dialog boxes contain scroll bars. This option applies only to HTML objects when you are using the DOM extension.</p>
OPT_SET_TARGET_MACHINE	Classic Agent	<p>STRING</p> <p>The IP address and port number to use for the target machine in distributed testing using the <code>SetOption</code> method. To set the target machine, type: <code>Agent.SetOption(OPT_SET_TARGET_MACHINE, < IPAddress >:< PortNumber >)</code> .</p> <p> Note: A colon must separate the IP address and the port number.</p> <p>To return the IP address and port number of the current target machine, type: <code>Agent.GetOption(OPT_SET_TARGET_MACHINE)</code></p>
OPT_SHOW_OUT_OF_VIEW	Classic Agent	<p>BOOLEAN</p> <p>TRUE to have the agent see a control not currently scrolled into view; FALSE to have the Agent consider an out-of-view window to be invisible. This option applies only to HTML</p>

Agent Option	Agent Supported	Description
OPT_SYNC_TIMEOUT	Open Agent	<p>objects when you are using the DOM extension.</p> <p>NUMBER</p> <p>Specifies the maximum time in seconds for an object to be ready.</p> <p> Note: When you upgrade from a Silk Test version prior to Silk Test 13.0, and you had set the OPT_XBROWSER_SYNC_TIMEOUT option, the Options dialog box will display the default value of the OPT_SYNC_TIMEOUT, although your timeout is still set to the value you have defined.</p>
OPT_TEXT_NEW_LINE	Classic Agent	<p>STRING</p> <p>The keys to type to enter a new line using the <code>SetMultiText</code> method of the <code>TextField</code> class. The default value is "<Enter>".</p>
OPT_TRANSLATE_TABLE	Classic Agent	<p>STRING</p> <p>Specifies the name of the translation table to use. If a translation DLL is in use, the <code>QAP_SetTranslateTable</code> entry point is called with the string specified in this option.</p>
OPT_TRIM_ITEM_SPACE	Classic Agent	<p>BOOLEAN</p> <p>TRUE to trim leading and trailing spaces from items on windows, or FALSE to avoid trimming spaces.</p>
OPT_USE_ANSICALL	Classic Agent	<p>BOOLEAN</p> <p>If set to TRUE, each following DLL function is called as ANSI. If set to FALSE, which is the default value, UTF-8 DLL calls are used. For single ANSI DLL calls you can also use the <code>ansicall</code> keyword.</p>
OPT_USE_SILKBEAN	Classic Agent	<p>BOOLEAN</p> <p>TRUE to enable the agent to interact with the SilkBean running on a UNIX machine.</p> <p>Default is FALSE.</p>
OPT_VERIFY_ACTIVE	Classic Agent Open Agent	<p>BOOLEAN</p> <p>TRUE to verify that windows are active before interacting with them;</p>


Agent Option	Agent Supported	Description
OPT_VERIFY_APPREADY	Classic Agent	<p>FALSE to not check. See Active and Enabled Statuses for information about how this option affects Silk Test Classic methods.</p> <p>Default is TRUE, except when running script statements that were recorded and are in a recording statement.</p>
OPT_VERIFY_CLOSED	Classic Agent	<p>BOOLEAN</p> <p>TRUE to synchronize the agent with the application under test. Calls to the agent will not proceed unless the application is ready.</p> <p>TRUE to verify that a window has closed. When FALSE, Silk Test Classic closes a window as usual, but does not verify that the window actually closed.</p> <p>Default is TRUE, except when running script statements that were recorded and are in a recording statement.</p>
OPT_VERIFY_COORD	Classic Agent	<p>BOOLEAN</p> <p>TRUE to check that coordinates passed to a method are inside the window before the mouse is pressed; FALSE to not check. Typically, you use the checking feature unless you need to be able to pass coordinates outside of the window, such as negative coordinates.</p> <p>If this option is set to TRUE and coordinates fall outside the window, Silk Test Classic raises the exception E_COORD_OUTSIDE_WINDOW.</p>
OPT_VERIFY_CTRLTYPE	Classic Agent	<p>BOOLEAN</p> <p>TRUE to check that objects are of the specified type before interacting with them; FALSE to not check.</p> <p>When TRUE, Silk Test Classic checks, for example, that an object that claims to be a listbox is actually a listbox. For custom objects, you must map them to the standard types to prevent the checking from signaling an exception, using the Silk Test Classic class map facility.</p> <p>Default is FALSE.</p>

Agent Option	Agent Supported	Description
OPT_VERIFY_ENABLED	Classic Agent	<p>BOOLEAN</p> <p>TRUE to verify that windows are enabled before interacting with them; FALSE to not check. For information about how this option affects various Silk Test Classic methods, see <i>Active and Enabled Statuses</i>.</p>
OPT_VERIFY_EXPOSED	Classic Agent	<p>BOOLEAN</p> <p>TRUE to verify that windows are exposed (that is, not covered, obscured, or logically hidden by another window) before interacting with them; FALSE to not check.</p> <p>Default is TRUE, except when running script statements that were recorded and are in a recording statement.</p>
OPT_VERIFY_RESPONDING	Classic Agent	<p>BOOLEAN</p> <p>Setting this option to FALSE suppresses "control not responding" errors.</p>
OPT_VERIFY_UNIQUE	<p>Classic Agent</p> <p>Open Agent</p>	<p>BOOLEAN</p> <p>TRUE to raise the E_WINDOW_NOT_UNIQUE exception upon encountering two or more windows with the same tag; FALSE to not raise the exception. When OPT_VERIFY_UNIQUE is FALSE, Silk Test Classic ignores the duplication and chooses the first window with that tag that it encounters.</p> <p>You can use a modified tag syntax to refer to a window with a non-unique tag, even when OPT_VERIFY_UNIQUE is TRUE. You can either include an index number after the object, as in myDialog("Cancel[2]"), or you can specify the window by including the text of a child that uniquely identifies the window, such as "myDialog/uniqueText/...", where the unique text is the tag of a child of that window.</p>
OPT_WAIT_ACTIVE_WINDOW	Open Agent	<p>NUMBER</p> <p>The number of seconds Silk Test Classic waits for a window to become active. If a window does not become</p>

Agent Option	Agent Supported	Description
OPT_WAIT_ACTIVE_WINDOW_RETRY	Open Agent	<p>active within the specified time, Silk Test Classic raises an exception.</p> <p>To require the Open Agent to check the active state of a window, set OPT_ENSURE_ACTIVE_WINDOW to TRUE.</p> <p>By default, OPT_WAIT_ACTIVE_WINDOW is set to 2 seconds.</p> <p>NUMBER</p> <p>The number of seconds Silk Test Classic waits for a window to become active before trying to verify again that the window is active.</p> <p>To require the Open Agent to retry the active state of an object, set OPT_ENSURE_ACTIVE_WINDOW to TRUE.</p> <p>By default, OPT_WAIT_ACTIVE_WINDOW_RETRY is set to 0.5 seconds.</p>
OPT_WINDOW_MOVE_TOLERANCE	Classic Agent	<p>INTEGER</p> <p>The number of pixels allowed for a tolerance when a moved window does not end up at the specified position.</p> <p>For some windows and GUIs, you cannot always move the window to the specified pixel. If the ending position is not exactly what was specified and the difference between the expected and actual positions is greater than the tolerance, Silk Test Classic raises an exception.</p> <p>On Windows, the tolerance can be set through the Control Panel, by setting the desktop window granularity option. If the granularity is zero, you can place a window at any pixel location. If the granularity is greater than zero, the desktop is split into a grid of the specified pixels in width, determining where a window can be placed. In general, the tolerance should be greater than or equal to the granularity.</p>
OPT_WINDOW_RETRY	Classic Agent Open Agent	NUMBER

Agent Option	Agent Supported	Description
		<p>The number of seconds Silk Test Classic waits between attempts to verify a window, if the window does not exist or is in the incorrect state. Silk Test Classic continues trying to find the window until the time specified with OPT_WINDOW_TIMEOUT is reached.</p> <p>The correct state of the window depends on various options. For example, Silk Test Classic might check whether a window is enabled, active, exposed, or unique, depending on the settings of the following options:</p> <ul style="list-style-type: none"> • OPT_VERIFY_ENABLED • OPT_VERIFY_ACTIVE • OPT_VERIFY_EXPOSED • OPT_VERIFY_UNIQUE
OPT_WINDOW_SIZE_TOLERANCE	Classic Agent	<p>INTEGER</p> <p>The number of pixels allowed for a tolerance when a resized window does not end at the specified size.</p> <p>For some windows and GUIs, you cant always resize the window to the particular size specified. If the ending size is not exactly what was specified and the difference between the expected and actual sizes is greater than the tolerance, Silk Test Classic raises an exception.</p> <p>On Windows, windows cannot be sized smaller than will fit comfortably with the menu bar.</p>
OPT_WINDOW_TIMEOUT	Classic Agent Open Agent	<p>NUMBER</p> <p>The number of seconds Silk Test Classic waits for a window to appear and be in the correct state. If a window does not appear within the specified timeout, Silk Test Classic raise an exception.</p> <p>The correct state of the window depends on various options. For example, Silk Test Classic might check whether a window is enabled, active, exposed, or unique, depending</p>

Agent Option	Agent Supported	Description
OPT_WPF_CHECK_DISPATCHER_FOR_IDLE	Open Agent	<p>on the settings of the following options:</p> <ul style="list-style-type: none"> • OPT_VERIFY_ENABLED • OPT_VERIFY_ACTIVE • OPT_VERIFY_EXPOSED • OPT_VERIFY_UNIQUE <p>BOOLEAN</p> <p>For some WPF applications the Silk Test synchronization might not work due to how certain controls are implemented, resulting in Silk Test Classic not recognising when the WPF application is idle. Setting this option to FALSE disables the WPF synchronization and prevents Silk Test Classic from checking the WPF dispatcher, which is the thread that controls the WPF application. Set this option to FALSE to solve synchronization issues with certain WPF applications. By default, this is TRUE.</p>
OPT_WPF_CUSTOM_CLASSES	Open Agent	<p>LIST OF STRING</p> <p>Specify the names of any WPF classes that you want to expose during recording and playback. For example, if a custom class called MyGrid derives from the WPF Grid class, the objects of the MyGrid custom class are not available for recording and playback. Grid objects are not available for recording and playback because the Grid class is not relevant for functional testing since it exists only for layout purposes. As a result, Grid objects are not exposed by default. In order to use custom classes that are based on classes that are not relevant to functional testing, add the custom class, in this case MyGrid, to the OPT_WPF_CUSTOM_CLASSES option. Then you can record, playback, find, verify properties, and perform any other supported actions for the specified classes.</p>
OPT_WPF_PREFILL_ITEMS	Open Agent	<p>BOOLEAN</p> <p>Defines whether items in a WPFItemsControl, like</p>

Agent Option	Agent Supported	Description
		<p>WPFComboBox or WPFListBox, are pre-filled during recording and playback. WPF itself lazily loads items for certain controls, so these items are not available for Silk Test Classic if they are not scrolled into view. Turn pre-filling on, which is the default setting, to additionally access items that are not accessible without scrolling them into view. However, some applications have problems when the items are pre-filled by Silk Test Classic in the background, and these applications can therefore crash. In this case turn pre-filling off.</p>
OPT_XBROWSER_FIND_HIDDEN_INPUT_FIELDS	Open Agent	<p>BOOLEAN</p> <p>Specifies whether to display hidden input fields, which are HTML fields for which the tag includes <code>type="hidden"</code>. The default value is TRUE.</p>
OPT_XBROWSER_SYNC_MODE	Open Agent	<p>STRING</p> <p>Configures the supported synchronization mode for HTML or AJAX. Using the HTML mode ensures that all HTML documents are in an interactive state. With this mode, you can test simple Web pages. If more complex scenarios with Java script are used, it might be necessary to manually script synchronization functions, such as <code>WaitForObject</code>, <code>WaitForProperty</code>, <code>WaitForDisappearance</code>, or <code>WaitForChildDisappearance</code>. Using the AJAX mode eliminates the need to manually script synchronization functions. By default, this value is set to AJAX.</p>
OPT_XBROWSER_SYNC_TIMEOUT	Open Agent	<p>NUMBER</p> <p>Specifies the maximum time in seconds for an object to be ready.</p> <p> Note: Deprecated. Use the option <code>OPT_SYNC_TIMEOUT</code> instead.</p>
OPT_XBROWSER_SYNC_EXCLUDE_URLS	Open Agent	<p>STRING</p>

Agent Option	Agent Supported	Description
		<p>Specifies the URLs of any services or web pages that you want to exclude during page synchronization. Some AJAX frameworks or browser applications use special HTTP requests, which are permanently open in order to retrieve asynchronous data from the server. These requests may let the synchronization hang until the specified synchronization timeout expires. To prevent this situation, either use the HTML synchronization mode or specify the URL of the problematic request in the Synchronization exclude list setting.</p> <p>Type the entire URL or a fragment of the URL, such as <code>http://test.com/timeService</code> or <code>timeService</code>. Separate entries by comma. For example:</p> <pre>Agent.SetOption(OPT_XBROWSER_SYNC_EXCLUDE_URLS, { "fpdownload.macromedia.com", "fpdownload.adobe.com", "download.microsoft.com" })</pre>

Setting the Default Agent

Silk Test Classic automatically assigns a default agent to your project or scripts. When you create a new project, the agent currently selected in the toolbar is the default agent. Silk Test Classic automatically starts the default agent when you open a project or create a new project. You can configure Silk Test Classic to automatically connect to the Open Agent or the Classic Agent by default.

To set the default agent, perform one of the following:

- Click **Options > Runtime** and set the default agent in the **Runtime Options** dialog box.
- Click the appropriate agent icon in the toolbar.

When you enable extensions, set the recovery system, configure the application, or record a test case, Silk Test Classic uses the default agent. When you run a test, Silk Test Classic automatically connects to the appropriate agent. Silk Test Classic uses the window declaration, locator, or `Find` or `FindAll` command to determine which agent to use.

Setting the Default Agent Using the Runtime Options Dialog Box

To set the default agent using the **Runtime Options** dialog box:



1. In the main menu, click **Options > Runtime**. The **Runtime Options** dialog box opens.

2. Select the agent that you want to use as the default from the **Default Agent** list box.
3. If you use the Classic Agent, select the type of network you want to use in the **Network** list box. If you select the Open Agent, TCP/IP is automatically selected.
4. If you use named agents, select the local agent name from the **Agent Name** list box. For instance, if your environment uses multiple agents or a port that uses a value other than the default, select the local agent.
5. Click **OK**.

When you record a test case, Silk Test Classic automatically uses the default agent.

Setting the Default Agent Using the Toolbar Icons

From the main toolbar, click one of the following icons to set the default agent:

-  to use the Classic Agent.
-  to use the Open Agent.

Connecting to the Default Agent

Typically, the default agent starts automatically when it is needed by Silk Test Classic. However, you can connect to the default agent manually if it does not start or to verify that it has started.

To connect to the default Agent, from the main menu, click **Tools > Connect to Default Agent**.

The command starts the Classic Agent or the Open Agent on the local machine, depending on which agent is specified as the default in the **Runtime Options** dialog box. If the Agent does not start within 30 seconds, a message is displayed. If the default Agent is configured to run on a remote machine, you must connect to it manually.

Creating a Script that Uses Both Agents

You can create a script that uses the Classic Agent and the Open Agent. Recording primarily depends on the default agent while replaying the script primarily depends on the window declaration of the underlying control. If you create a script that does not use window declarations, the default agent is used to replay the script.

1. Set the default agent to the Classic Agent.
2. In the **Basic Workflow** bar, enable extensions for the application automatically.
3. In the **Basic Workflow** bar, click **Record Testcase** and record your test case.
4. When prompted, click **Paste to Editor** and then click **Paste testcase and update window declaration(s)**.
5. Click **OK**. The frame now contains window declarations from the Classic Agent.
6. Click **File > Save** to save the test case.
7. Type a name for the file into the **File name** field and click **Save**.
8. Set the default agent to the Open Agent.
9. Click **Options > Application Configurations**. The **Edit Application Configurations** dialog box opens.
10. Click **Add**.

The **Select Application** dialog box opens.

11. Configure a standard or Web site test configuration.
12. Click **OK**.
13. Click **Record Testcase** in the **Basic Workflow** bar and record your test case.

14. When prompted, click **Paste to Editor** and then click **Paste testcase and update window declaration(s)**. The frame now contains window declarations from both the Classic Agent and the Open Agent. Silk Test Classic automatically detects which agent is required for each test based on the window declaration and changes the agent accordingly.
15. Click **File > Save** to save the test case.
16. Click **Run Testcase** in the **Basic Workflow** bar to replay the test case. Silk Test Classic automatically recognizes which agent to use based on the underlying window declarations.


You can also use the function `Connect([sMachine, sAgentType])` in a script to connect a machine explicitly with either the Classic Agent or the Open Agent. Using the connect function changes the default agent temporarily for the current test case, but it does not change the default agent of your project. However, this does not override the agent that is used for replay, which is defined by the window declaration.

Overview of Record Functionality Available for the Silk Test Agents

The Open Agent provides the majority of the same record capabilities as the Classic Agent and the same replay capabilities.

The following table lists the record functionality available for each Silk Test agent.

Record Command	Classic Agent	Open Agent
Window Declarations	Supported	Supported
Application State	Supported	Supported
Testcase	Supported	Supported
Actions	Supported	Supported
Window Identifiers	Supported	Not Supported
Window Locations	Supported	Not Supported
Window Locators	Not Supported	Supported
Class/Scripted	Supported	Not Supported
Class/Accessibility	Supported	Not Supported
Method	Supported	Not Supported
Defined Window	Supported	Not Supported

 **Note:** Silk Test Classic determines which agent to use by detecting whether a locator or `Find` or `FindAll` command is used. If a locator or `Find` or `FindAll` command is present, Silk Test Classic uses the Open Agent. As a result, you do not need to record window declarations for the Open Agent. For calls that use window declarations, the agent choice is made based on the presence or absence of the locator keyword and on the presence or absence of `TAG_IS_OPEN_AGENT` in a tag or multitag. When a window declaration contains both locators and tags and either could be used for resolving the window, check or uncheck the **Prefer Locator** check box in the **General Options** dialog box to determine which method is used.

Setting the Window Timeout Value to Prevent Window Not Found Exceptions

The window timeout value is the number of seconds Silk Test Classic waits for a window to display. If the window does not display within that period, the Window not found exception is raised. For example, loading an Apache Flex application and initializing the Apache Flex automation framework may take some time, depending on the machine on which you are testing and the complexity of your Apache Flex application. In this case, setting the Window timeout value to a higher value enables your application to fully load.

If you suspect that Silk Test Classic is not waiting long enough for a window to display, you can increase the window timeout value in the following ways:

- Change the window timeout value on the **Timing** tab of the **Agent Options** dialog box.
- Manually add a line to the script.

If the window is on the screen within the amount of time specified in the window timeout, the tag for the object might be the problem.

Manually Setting the Window Timeout Value

In some cases, you may want to increase the window timeout value for a specific test, rather than for all tests in general. For example, you may want to increase the timeout for Flex application tests, but not for browser tests.

1. Open the test script.
2. Add the following to the script: `Agent.SetOption (OPT_WINDOW_TIMEOUT, numberOfSeconds).`

Setting the Window Timeout Value in the Agent Options Dialog Box

To change the window timeout value in the **Agent Options** dialog box:

1. Click **Options > Agent**.
2. Click the **Timing** tab.
3. Type the value into the **Window timeout** text box.

The value should be based on the speed of the machine, on which you are testing, and the complexity of the application that you are testing. By default, this value is set to 5 seconds. For example, loading and initializing complex Flex applications generally requires more than 5 seconds.

4. Click **OK**.

Configuring Open Agent Port Numbers

Typically, you do not have to configure port numbers manually. The information service handles port configuration automatically. Use the default port of the information service to connect to the agent. Then, the information service forwards communication to the port that the agent uses. However, if you have a port number conflict or an issue with a firewall, you must configure the port number for that machine or for the information service.

The default port of the information service is 22901. When you can use the default port, you can type `hostname` without the port number for ease of use. If you do specify a port number, ensure that it matches the value for the default port of the information service or one of the additional information service ports. Otherwise, communication will fail.

After changing the port number, restart the Open Agent, Silk Test Classic, Silk Test Recorder, and the application that you want to test.

Configuring the Port to Connect to the Information Service

Before you begin this task, stop the Silk Test Open Agent.

This functionality is supported only if you are using the Open Agent.

Typically, you do not have to configure port numbers manually. The information service handles port configuration automatically. Use the default port of the information service to connect to the Agent. Then, the information service forwards communication to the port that the Agent uses.

The default port of the information service is 22901. When you can use the default port, you can type `hostname` without the port number for ease of use. If you do specify a port number, ensure that it matches the value for the default port of the information service or one of the additional information service ports. Otherwise, communication will fail.

If necessary, you can change the port number that all clients use to connect to the information service.

1. Navigate to the `infoservice.properties.sample` file and open it.

- On a Windows system, this file is located in `C:\ProgramData\Application Data\Silk\SilkTest\conf`, where “`C:\ProgramData`” is equivalent to the content of the `ALLUSERSPROFILE` environment variable, which is set by default on Windows systems.
- On macOS, this file is located in `/Users/<user>/.silk/silktest/conf`.

This file contains commented text and sample alternate port settings.

2. Change the value for the appropriate port.

Typically, you configure the information service port settings to avoid problems with a firewall by forcing communication on a specific port.

Port numbers can be any number from 1 to 65535.

- `infoservice.default.port` – The default port where the information service runs. By default, this port is set to 22901.

3. Save the file as `infoservice.properties`.

4. Restart the Open Agent, the Silk Test client, and the application that you want to test.

Open Agent Port Numbers

When the Open Agent starts, a random available port is assigned to Silk Test Classic, Silk Test Recorder, and the application that you are testing. The port numbers are registered on the information service. Silk Test Classic and Silk Test Recorder contact the information service to determine the port to use to connect to the Open Agent. The information service communicates the appropriate port, and Silk Test Classic or the Silk Test Recorder connect to that port. Communication runs directly between Silk Test Classic or the Silk Test Recorder and the agent.

By default, the Open Agent communicates with the information service on port 22901. You can configure additional ports for the information service as alternate ports that work when the default port is not available. By default, the information service uses ports 2966, 11998, and 11999 as alternate ports.

Typically, you do not have to configure port numbers manually. However, if you have a port number conflict or an issue with a firewall, you must configure the port number for that machine or for the information service. You can use a different port number for a single machine or you can use the same available port number for all your machines.

Stopping the Open Agent After Test Execution

You can stop the Open Agent from a script, to ensure that the agent does not continue running after the end of the test execution.

1. Open or create a script that is executed when the test execution is finished.
For example, open an existing script that is used for cleanup after test execution.
2. Add the `ShutDown` method to the script.



Note: The Open Agent will restart as soon as the agent is required by another script.

Basic Workflow for the Open Agent

The **Basic Workflow** bar guides you through the process of creating a test case. To create and execute a test case, click each icon in the workflow bar to perform the relevant procedures. The procedures and the appearance of the workflow bar differ depending on whether your test uses the Open Agent or the Classic Agent.

The **Basic Workflow** bar is displayed by default. You can display it or hide it by checking and un-checking the **Workflows > Basic** check box. If your test uses both the Open Agent and the Classic Agent, the **Basic Workflow** bar changes when you switch between the agents.

When you use the Open Agent, the **Basic Workflow** uses dynamic object recognition to record and replay test cases that use XPath queries to find and identify objects.

Creating a New Project

You can create a new project and add the appropriate files to the project, or you can have Silk Test Classic automatically create a new project from an existing file.

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by opening Silk Test Classic and clicking **Options > Save New Options Set**. You can add the options set to your project.

To create a new project:

1. In Silk Test Classic, click **File > New Project**, or click **Open Project > New Project** on the basic workflow bar. the **Create Project** dialog box appears.
2. Type a unique name for the project into the **Project Name** field.
If the project name already exists, you are prompted to specify a different project name or overwrite the existing project.
Project files with a `.vtp` (Verify Test Project) extension, `projectname.vtp`, and a `projectname.ini` are created in the directory that you specify in the **Save in** text box.
3. *Optional:* Type a description for the project into the **Description** field.
After you create the project, you can view the project description by right-clicking the `projectname` folder on the **Files** or **Global** tabs or clicking **Project > Project Description**.
4. Type the location at which you want to save the project into the **Save in** field.
You can also click **Browse** and specify the folder in which you want to save your project. The default location is `C:\Users\\Documents\Silk Test Classic Projects`.
5. Click **OK**. Silk Test Classic creates a `<Project name>` folder within the specified directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension `.ini` files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project. Silk Test Classic then creates your project and displays nodes on the **Files** and **Global** tabs for the files and resources associated with this project.
6. Perform one of the following steps:
 - If your test uses the Open Agent, configure the application to set up the test environment.
 - If your test uses the Classic Agent, enable the appropriate extensions to test your application.

Configuring Applications

When you configure an application, Silk Test Classic automatically creates a base state for the application. An application's base state is the known, stable state that you expect the application to be in before each test begins execution, and the state the application can be returned to after each test has ended execution.

Silk Test Classic has slightly different procedures depending on which type of application you are configuring:

- A standard application, which is an application that does not use a web browser, for example a Windows application or a Java SWT application.
- A web application, which is an application that uses a web browser, for example a web page, a web-based application on a mobile device, or an Apache Flex application.
- A mobile native application, which is a non web-based application on a mobile device.

Recording Test Cases for Standard and Web Applications



Note: Before you can record a test case, you have to configure the application that you want to test.

This functionality is supported only if you are using the Open Agent.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. With this approach, you combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.

1. Click **Record Testcase** on the basic workflow bar. If the workflow bar is not visible, click **Workflows > Basic** to enable it. The **Select Test Type** dialog box opens.
2. Select **4Test Testcase**.
3. Click **OK**. The **Record Testcase** dialog box opens.
4. Type the name of your test case into the **Testcase name** field.

Test case names are not case sensitive; they can have any length and they can consist of any combination of alphabetic characters, numerals, and underscore characters.

5. From the **Application State** list box, select **DefaultBaseState** to have the built-in recovery system restore the default base state before the test case begins executing.

- If you choose *DefaultBaseState* as the application state, the test case is recorded in the script file as `testcase testcase_name ()`.
- If you choose another application state, the test case is recorded as `testcase testcase_name () appstate appstate_name`.

6. Click **Start Recording**. Silk Test Classic performs the following actions:

- Closes the **Record Testcase** dialog box.
- Starts your application, if it was not already running. If you have not configured the application yet, the **Select Application** dialog box opens and you can select the application that you want to test.
- Removes the editor window from the display.
- Displays the **Recording** window.
- Waits for you to take further action.

7. In the application under test, perform the actions that you want to test.

For information about the actions available during recording, see *Actions Available During Recording*.

8. To stop recording, click **Stop** in the **Recorder**. Silk Test Classic displays the **Record Testcase** dialog box, which contains the code that has been recorded for you.
9. To resume recording your interactions, click **Resume Recording**.
10. To add the recorded interactions to a script, click **Paste to Editor** in the **Record Testcase** window. If you have interacted with objects in your application that have not been identified in your include files, the **Update Files** dialog box opens.
11. Perform one of the following steps:
 - Click **Paste testcase and update window declaration(s)** and then click **OK**. In most cases, you want to choose this option.
 - Choose **Paste testcase only** and then click **OK**. This option does not update the window declarations in the INC file when it pastes the script to the Editor. If you previously recorded the window declarations related to this test case, choose this option.
12. Click **OK**. Silk Test Classic adds the recorded actions to the specified script file, and opens the file in the editor.

Recording Test Cases for Mobile Applications

This functionality is supported only if you are using the Open Agent.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. With this approach, you combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.

1. Click **Record Testcase** on the basic workflow bar. If the workflow bar is not visible, click **Workflows > Basic** to enable it. The **Select Test Type** dialog box opens.
2. Select **4Test Testcase**.
3. Click **OK**. The **Record Testcase** dialog box opens.
4. Type the name of your test case into the **Testcase name** field.

Test case names are not case sensitive; they can have any length and they can consist of any combination of alphabetic characters, numerals, and underscore characters.
5. From the **Application State** list box, select **DefaultBaseState** to have the built-in recovery system restore the default base state before the test case begins executing.
 - If you choose *DefaultBaseState* as the application state, the test case is recorded in the script file as `testcase testcase_name ()`.
 - If you choose another application state, the test case is recorded as `testcase testcase_name () appstate appstate_name`.
6. Click **Start Recording**. Silk Test Classic performs the following actions:
 - Closes the **Record Testcase** dialog box.
 - Starts your application, if it was not already running. If you have not configured the application yet, the **Select Application** dialog box opens and you can select the application that you want to test.
 - Removes the editor window from the display.
 - Displays the **Recording** window.
 - Waits for you to take further action.
7. On the device or the Emulator, interact with your native mobile application, driving it to the state that you want to test.
8. In the **Recording** window, perform the actions that you want to record.
 - a) Click on the object with which you want to interact. Silk Test Classic performs the default action for the object. If there is no default action, or if you have to insert text or specify parameters, the **Choose Action** dialog box opens.

- b) *Optional*: To choose an action for an object, which might not be the default action, right-click on the object. The **Choose Action** dialog box opens.
- c) *Optional*: If the action has parameters, type the parameters into the parameter fields. Silk Test Classic automatically validates the parameters.
- d) Click **OK** to close the **Choose Action** dialog box. Silk Test Classic adds the action to the recorded actions and replays it on the mobile device or emulator.

For information about how to record an interaction with a mobile device, see *Interacting with a Mobile Device*.

9. To verify an image or a property of a control during recording, click `Ctrl+Alt`.

For additional information, see [Adding a Verification to a Script while Recording](#).

10. *Optional*: To interact with an object that is currently not visible in the **Recording** window, use the **Hierarchy View**:

- a) Click **Toggle Hierarchy View**. The **Hierarchy View** opens.
- b) In the object tree, right-click on the object on which you want to perform an action.
- c) Click **Add New Action**. The **Choose Action** dialog box opens.
- d) Proceed as with any other action.

For example, to open the main menu of the device or emulator, right-click on the *MobileDevice* object in the object tree and select the action `PressMenu()`.

11. To pause the recording of interactions with the application, for example to move the application into a different state, click **Pause**.

12. To resume recording interactions, click **Start**.

13. To add the recorded interactions to a script, click **Stop**. If you have interacted with objects in your application that have not been identified in your include files, the **Update Files** dialog box opens.

14. Perform one of the following steps:

- Click **Paste testcase and update window declaration(s)** and then click **OK**. In most cases, you want to choose this option.
- Choose **Paste testcase only** and then click **OK**. This option does not update the window declarations in the INC file when it pastes the script to the Editor. If you previously recorded the window declarations related to this test case, choose this option.

Running a Test Case

When you run a test case, Silk Test Classic interacts with the application by executing all the actions you specified in the test case and testing whether all the features of the application performed as expected.

Silk Test Classic always saves the suite, script, or test plan before running it if you made any changes to it since the last time you saved it. By default, Silk Test Classic also saves all other open modified files whenever you run a script, suite, or test plan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog box.

1. Make sure that the test case that you want to run is in the active window.

2. Click **Run Testcase** on the **Basic Workflow** bar.

If the workflow bar is not visible, choose **Workflows > Basic** to enable it.

Silk Test Classic displays the **Run Testcase** dialog box, which lists all the test cases contained in the current script.

3. Select a test case and specify arguments, if necessary, in the **Arguments** field.

Remember to separate multiple arguments with commas.

4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

- BaseStateExecutionFinished
- Connecting
- Verify
- Exists
- Is
- Get
- Set
- Print
- ForceActiveXEnum
- Wait
- Sleep

5. To view results using the TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.
6. Click **Run**. Silk Test Classic runs the test case and generates a results file.
For the Classic Agent, multiple tags are supported. If you are running test cases using other agents, you can run scripts that use declarations with multiple tags. To do this, check the **Disable Multiple Tag Feature** check box in the **Agent Options** dialog box on the **Compatibility** tab. When you turn off multiple-tag support, 4Test discards all segments of a multiple tag except the first one.
7. *Optional:* If necessary, you can press both **Shift** keys at the same time to stop the execution of the test.

Viewing Test Results

Whenever you run tests, a results file is generated which indicates how many tests passed and how many failed, describes why tests failed, and provides summary information.

1. Click **Explore Results** on the **Basic Workflow** or the **Data Driven Workflow** bars.
2. On the **Results Files** dialog box, navigate to the file name that you want to review and click **Open**.


By default, the results file has the same name as the executed script, suite, or test plan. To review a file in the TrueLog Explorer, open a `.xlg` file. To review a results file, open a `.res` file.

Migrating from the Classic Agent to the Open Agent

This section includes several useful topics that explain the differences between the Classic Agent and the Open Agent. If you plan to migrate from testing using the Classic Agent to the Open Agent, review this information to learn how to migrate your existing assets including window declarations and scripts.

Differences for Agent Options Between the Silk Test Agents

Before you migrate existing Classic Agent scripts to the Open Agent, review the Agent Options listed below to determine if any additional action is required to facilitate the migration.

Classic Agent Option	Action for Open Agent
OPT_AGENT_CLICKS_ONLY	Option not needed.  Note: Use OPT_REPLAY_MODE for switching between high-level (API) clicks and low-level clicks.
OPT_CLOSE_MENU_NAME	Not supported by Open Agent.
OPT_COMPATIBLE_TAGS	Option not needed.
OPT_COMPRESS_WHITESPACE	Not supported by Open Agent.
OPT_DROPDOWN_PICK_BEFORE_GET	Option not needed. The Open Agent performs this action by default during replay.
OPT_EXTENSIONS	Option not needed.
OPT_GET_MULTITEXT_KEEP_EMPTY_LINES	Not supported by Open Agent.
OPT_KEYBOARD_LAYOUT	Not supported by Open Agent.
OPT_MENU_INVOKE_POPUP	No action. Pop-up menu handling using the Open Agent does not need such an option.
OPT_MENU_PICK_BEFORE_GET	Option not needed.
OPT_NO_ICONIC_MESSAGE_BOXES	Option not needed.
OPT_PLAY_MODE	Option not needed.
OPT_RADIO_LIST	Open Agent always sees <code>RadioList</code> items as individual objects.
OPT_REL1_CLASS_LIBRARY	Obsolete option.
OPT_REQUIRE_ACTIVE	Use the option OPT_ENSURE_ACTIVE instead.
OPT_SCROLL_INTO_VIEW	Option not needed. Open Agent only requires scrolling into view for low-level replay. By default, high-level replay is used, so no scrolling needs to be performed. However, <code>CaptureBitmap</code> never scrolls an object into view.
OPT_SET_TARGET_MACHINE	Option not needed.

Classic Agent Option	Action for Open Agent
OPT_SHOW_OUT_OF_VIEW	Option not needed. Out-of-view objects are always recognized.
OPT_TEXT_NEW_LINE	Option not needed. The Open Agent always uses Enter to type a new line.
OPT_TRANSLATE_TABLE	Not supported by Open Agent.
OPT_TRAP_FAULTS	Fault trap is no longer active.
OPT_TRAP_FAULTS_FLAGS	Fault trap is no longer active.
OPT_TRIM_ITEM_SPACE	Option not needed. If required, use a * wildcard instead.
OPT_USE_ANSICALL	Not supported by Open Agent.
OPT_USE_SILKBEAN	SilkBean is not supported on the Open Agent.
OPT_VERIFY_APPREADY	Option not needed. The Open Agent performs this action by default.
OPT_VERIFY_CLOSED	Option not needed. The Open Agent performs this action by default.
OPT_VERIFY_COORD	Option not needed. The Open Agent does not typically check for native input in order to allow clicking outside of an object.
OPT_VERIFY_CTRLTYPE	Option not needed.
OPT_VERIFY_EXPOSED	Option not needed. The Open Agent performs this action when it sets a window to active. OPT_ENSURE_ACTIVE_OBJECT_DEF should yield the same result.
OPT_VERIFY_RESPONDING	Option not needed.
OPT_WINDOW_MOVE_TOLERANCE	Option not needed.

Differences in Object Recognition Between the Silk Test Agents

When recording and executing test cases, the Classic Agent uses the keywords `tag` or `multitag` in a window declaration to uniquely identify an object in the test application. The `tag` is the actual name, as opposed to the identifier, which is the logical name.

When using the Open Agent, you typically use dynamic object recognition with a `Find` or `FindAll` function and an XPath query to locate objects in your test application. To make calls that use window declarations using the Open Agent, you must use the keyword `locator` in your window declarations. Similar to the `tag` or `multitag` keyword, the `locator` is the actual name, as opposed to the identifier, which is the logical name. This similarity facilitates a smooth transition of legacy window declarations, which use the Classic Agent, to dynamic object recognition, which leverages the Open Agent.

The following sections explain how to migrate the different tag types to valid locator strings.

Caption

Classic Agent `tag "<caption string>"`

Open Agent `locator "//<class name>[@caption='<caption string>']"`



Note: For convenience, you can use shortened forms for the XPath locator strings. Silk Test Classic automatically expands the syntax to use full XPath strings when you run a script.

You can omit:

- The hierarchy separator, “./”. Silk Test Classic defaults to “/”.
- The class name. Silk Test Classic defaults to the class name of the window that contains the locator.
- The surrounding square brackets of the attributes, “[]”.
- The “@caption=” if the XPath string refers to the caption.



Note: Classic Agent removes ellipses (...) and ampersands (&) from captions. Open Agent removes ampersands, but not ellipses.

Example

Classic Agent:

```
CheckBox CaseSensitive  
tag "Case sensitive"
```

Open Agent:

```
CheckBox CaseSensitive  
locator "//CheckBox[@caption='Case sensitive']"
```

Or, if using the shortened form:

```
CheckBox CaseSensitive  
locator "Case sensitive"
```

Prior text

Classic Agent tag "^Find What:"

Open Agent locator "//<class name>[@priorlabel='Find What:']"



Note: Only available for Windows API-based and Java Swing applications. For other technology domains, use the **Locator Spy** to find an alternative locator.

Index

Classic Agent tag "#1"

Open Agent Record window locators for the test application. The Classic Agent creates index values based on the position of controls, while the Open Agent uses the controls in the order provided by the operating system. As a result, you must record window locators to identify the current index value for controls in the test application.

Window ID

Classic Agent tag "\$1041"

Open Agent locator "//<class name>[@windowid='1041']"

Location

Classic Agent tag "@(57,75)"

Open Agent not supported



Note: If you have location tags in your window declarations, use the **Locator Spy** to find an alternative locator.

Multitag

Classic Agent multitag "Case sensitive" "\$1011"

Open Agent locator "//CheckBox[@caption='Case sensitive' or @windowid='1011']" 'parent' statement

No changes needed. Multitag works the same way for the Open Agent.


Differences in the Classes Supported by the Silk Test Agents

The Classic Agent and the Open Agent differ slightly in the types of classes that they support. These differences are important if you want to manually script your test cases. Or, if you are testing a single test environment with both the Classic Agent and the Open Agent. Otherwise, the Open Agent provides the majority of the same record capabilities as the Classic Agent and the same replay capabilities.

Windows-based applications

Both Agents support testing Windows API-based client/server applications. The Open Agent classes, functions, and properties differ slightly from those supported on the Classic Agent for Windows API-based client/server applications.

Classic Agent	Open Agent
AnyWin	AnyWin
AgentClass (Agent)	AgentClass (Agent)
CheckBox	CheckBox
ChildWin	<no corresponding class>
ClipboardClass (Clipboard)	ClipboardClass (Clipboard)
ComboBox	ComboBox
Control	Control
CursorClass (Cursor)	CursorClass (Cursor)
CustomWin	CustomWin
DefinedWin	<no corresponding class>
DesktopWin (Desktop)	DesktopWin (Desktop)
DialogBox	DialogBox
DynamicText	<no corresponding class>
Header	HeaderEx
ListBox	ListBox
ListView	ListViewEx
MainWin	MainWin
Menu	Menu
MenuItem	MenuItem
MessageBoxClass	<no corresponding class>

Classic Agent	Open Agent
MoveableWin	MoveableWin
PageList	PageList
PopupList	ComboBox
PopupMenu	<no corresponding class>
PopupStart	<no corresponding class>
PopupSelect	<no corresponding class>
PushButton	PushButton
RadioButton	 Note: Items in Radiolists are recognized as RadioButtons on the CA. OA only identifies all of those buttons as RadioList.
RadioList	RadioList
Scale	Scale
ScrollBar	ScrollBar, VerticalScrollBar, HorizontalScrollBar
StaticText	StaticText
StatusBar	StatusBar
SysMenu	<no corresponding class>
Table	TableEx
TaskbarWin (Taskbar)	<no corresponding class>
TextField	TextField
ToolBar	ToolBar Additionally: PushToolItem, CheckBoxToolItem
TreeView, TreeViewEx	TreeView
UpDown	UpDownEx

The following core classes are supported on the Open Agent only:

- CheckBoxToolItem
- DropDownToolItem
- Group
- Item
- Link
- MonthCalendar
- Pager
- PushToolItem
- RadioListToolItem
- ToggleButton
- ToolItem

Web-based Applications

Both Agents support testing Web-based applications. The Open Agent classes, functions, and properties differ slightly from those supported on the Classic Agent for Windows API-based client/server applications.

Classic Agent	Open Agent
Browser	BrowserApplication
BrowserChild	BrowserWindow
HtmlCheckBox	DomCheckBox
HtmlColumn	<no corresponding class>
HtmlComboBox	<no corresponding class>
HtmlForm	DomForm
HtmlHeading	<no corresponding class>
HtmlHidden	<no corresponding class>
HtmlImage	<no corresponding class>
HtmlLink	DomLink
HtmlList	<no corresponding class>
HtmlListBox	DomListBox
HtmlMarquee	<no corresponding class>
HtmlMeta	<no corresponding class>
HtmlPopupList	DomListBox
HtmlPushButton	DomButton
HtmlRadioButton	DomRadioButton
HtmlRadioList	<no corresponding class>
HtmlTable	DomTable
HtmlText	<no corresponding class>
HtmlTextField	DomTextField
XmlNode	<no corresponding class>
Xul* Controls	<no corresponding class>



Note: The `DomElement` class of the Open Agent enables you to access any element on an HTML page. If the Open Agent has no class associated with a specific class supported on the Classic Agent, you can use the `DomElement` class to access the controls in the class.

Java AWT/Swing Applications

Both Agents support testing Java AWT/Swing applications. The Open Agent classes, functions, and properties differ slightly from those supported on the Classic Agent for Windows API-based client/server applications.

Classic Agent	Open Agent
JavaApplet	AppletContainer
JavaDialogBox	AWTDialog, JDialog
JavaMainWin	AWTFrame, JFrame
JavaAwtCheckBox	AWTCheckBox

Classic Agent	Open Agent
JavaAwtListBox	AWTList
JavaAwtPopupList	AWTChoice
JavaAwtPopupMenu	<no corresponding class>
JavaAwtPushButton	AWTPushButton
JavaAwtRadioButton	AWTRadioButton
JavaAwtRadioList	<no corresponding class>
JavaAwtScrollBar	AWTScrollBar
JavaAwtStaticText	AWTLabel
JavaAwtTextField	AWTTextField, AWTextArea
JavaJFCCheckBox	JCheckBox
JavaJFCCheckBoxMenuItem	JCheckBoxMenuItem
JavaJFCChildWin	<no corresponding class>
JavaJFCComboBox	JComboBox
JavaJFCImage	<no corresponding class>
JavaJFCListBox	JList
JavaJFCMenu	JMenu
JavaJFCMenuItem	JMenuItem
JavaJFCPageList	JTabbedPane
JavaJFCPopupList	JList
JavaJFCPopupMenu	JPopupMenu
JavaJFCProgressBar	JProgressBar
JavaJFCPushButton	JButton
JavaJFCRadioButton	JRadioButton
JavaJFCRadioButtonMenuItem	JRadioButtonMenuItem
JavaJFCRadioList	<no corresponding class>
JavaJFCScale	JSlider
JavaJFCScrollBar	JScrollBar, JHorizontalScrollBar, JVerticalScrollBar
JavaJFCSeparator	JComponent
JavaJFCStaticText	JLabel
JavaJFCTable	JTable
JavaJFCTextField	JTextField, JTextArea
JavaJFCToggleButton	JToggleButton
JavaJFCToolBar	JToolBar
JavaJFCTreeView	JTree

Classic Agent	Open Agent
JavaJFCUpDown	JSpinner

Java SWT/RCP Applications

Only the Open Agent supports testing Java SWT/RCP-based applications. For a list of the classes, see *Supported SWT Widgets for the Open Agent*.

Differences in the Parameters Supported by the Silk Test Agents

The Classic Agent and the Open Agent differ slightly in the function parameters that they support. These differences are important if you want to manually script your test cases. Or, if you are testing a single test environment with both the Classic Agent and the Open Agent. Otherwise, the Open Agent provides the majority of the same record capabilities as the Classic Agent and the same replay capabilities.

For some parameters, the Open Agent uses a hard-coded default value internally. If one of these parameters is set in a 4Test script, the Open Agent ignores the value and uses the value listed here.

Function	Parameter	Classic Agent Value	Open Agent Value
AnyWin::PressKeys/ ReleaseKeys	nDelay	Any number.	0
AnyWin::PressKeys/ ReleaseKeys	sKeys	More than one key is supported.	Only one key is supported. The first key is used and the remaining keys are ignored. For example <code>MainWin.PressKeys("<Shift><Left>")</code> will only press the Shift key. To press both keys, specify <code>MainWin.PressKeys("<Shift>")</code> or <code>MainWin.PressKeys("<Left >")</code> .
AnyWin::TypeKeys	sEvents	Keystrokes to type or mouse buttons to press.	The Open Agent supports keystrokes only.
AnyWin::GetChildren	bInvisible	TRUE or FALSE.	FALSE.
AnyWin::GetChildren	bNoTopLevel	TRUE or FALSE.	FALSE.
TextField::GetFontName	iLine	The Classic Agent recognizes this parameter.	The Open Agent ignores this parameter.
AnyWin::GetCaption	bNoStaticText	TRUE or FALSE.	FALSE.
AnyWin::GetCaption, Control::GetPriorStatic	bRawMode	TRUE or FALSE.	FALSE. However, the returned strings include trailing and leading spaces, but ellipses, accelerators, and hot keys are removed.
PageList::GetContents/ GetPageName	bRawMode	TRUE or FALSE.	FALSE. However, the returned strings include trailing and leading spaces, ellipses, and hot keys but accelerators are removed.

Function	Parameter	Classic Agent Value	Open Agent Value
AnyWin::Click/ DoubleClick/ MoveMouse/ MultiClick/ PressMouse/ ReleaseMouse, PushButton::Click	bRawEvent	The Classic Agent recognizes this parameter.	The Open Agent ignores this value.

Overview of the Methods Supported by the Silk Test Agents

The `winclass.inc` file includes information about which methods are supported for each Silk Test Classic Agent. The following 4Test keywords indicate Agent support:

- supported_ca** Supported on the Classic Agent only.
- supported_oa** Supported on the Open Agent only.

Standard 4Test methods, such as `AnyWin::GetCaption()`, can be marked with one of the preceding keywords. A method that is marked with the `supported_ca` or `supported_oa` keyword can only be executed successfully on the corresponding Agent. Methods that do not have a keyword applied will run on both Agents.

To find out which methods are supported on each Agent, open the `.inc` file, for instance `winclass.inc`, and verify whether the `supported_ca` or `supported_oa` keyword is applied to it.

Classic Agent

Certain functions and methods run on the Classic Agent only. When these are recorded and replayed, they default to the Classic Agent automatically. You can use these in an environment that uses the Open Agent. Silk Test Classic automatically uses the appropriate Agent. The functions and methods include:

- C data types for use in calling functions in DLLs.
- `ClipboardClass` methods.
- `CursorClass` methods.
- Certain SYS functions.

SYS Functions Supported by the Open Agent and the Classic Agent

The Classic Agent supports all SYS functions. The Open Agent supports all SYS functions with the exception of `SYS_GetMemoryInfo`. `SYS_GetMemoryInfo` defaults to the Classic Agent when a script is executed.

The following SYS functions behave differently depending on the agent that is used:

- | SYS Function | Description |
|-----------------------------|--|
| SYS_GetRegistryValue | With the Classic Agent, <code>SYS_GetRegistryValue</code> returns an incorrect value when a binary value is used. Use the Open Agent with <code>SYS_GetRegistryValue</code> to avoid this issue. |

SYS Function	Description
SYS_FileSetPointer	When setting the pointer after the end of the file, the Open Agent does not throw an exception, while the Classic Agent does throw an exception.
SYS_IniFileGetValue	The Open Agent does not allow the] character to be part of a section name, while the Classic Agent does allow it. Also, with the Open Agent, = must not be part of a key name. The Classic Agent allows = to be part of a key name, but produces incorrect results.



Note: Error messages and exceptions might differ between the Open Agent and the Classic Agent.

Silk Test Classic Projects

Silk Test Classic projects organize all the resources associated with a test set and present them visually in the **Project Explorer**, making it easy for you to see your test environment, and to manage it and work within it.

Silk Test Classic projects store relevant information about your project, including the following:

- References to all the resources associated with a test set, such as the following:
 - Data.
 - Frame files.
 - Include files.
 - `.ini` files.
 - Keyword-driven tests.
 - Option sets.
 - Results.
 - Script files.
 - Test plan files.
- Configuration information.
- Editor settings.
- Data files for attributes and queries.

All of this information is stored at the project level, meaning that once you add the appropriate files to your project and configure it once, you might never need to do it again. Switching among projects is easy - since you need to configure the project only once, you can simply open the project and run your tests.

When you create a new project, Silk Test Classic automatically uses the agent that is selected in the toolbar.

Each project is a unique testing environment

By default, new projects do not contain any settings, such as enabled extensions, class mappings, or agent options. If you want to retain the settings from your current test set, save them as a options set by opening Silk Test Classic and clicking **Options > Save New Options Set**. You can include the options set when you create your project. You can create a project manually or you can let Silk Test Classic automatically generate a project for you, based on existing files that you specify.



Note: To optimally use the functionality that Silk Test Classic provides, create an individual project for each application that you want to test, except when testing multiple applications in the same test.

Storing Project Information

Silk Test Classic stores project-related information in the following project files:

<code>projectname.vtp</code>	The project file has a Verify Test Project (<code>.vtp</code>) extension and is organized as an <code>.ini</code> file. It stores the names and locations of files used by the project.
<code>projectname.ini</code>	The project initialization file, similar to the <code>partner.ini</code> file, stores information about options sets, queries, and other resources included in your project.

SilkTestClassic.ini A user-specific initialization file that stores user-specific information about the location of the last projects, the size of the project history, and the location of the current project.

These files are created in the `projectname` folder. When you create your project, Silk Test Classic prompts you to store your project in the default location `C:\Users\\Documents\Silk Test Classic Projects`. Silk Test Classic creates a `<Project name>` folder within the specified directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension `.ini` files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project.

When you export a project, the default location is the project directory.



Note: The extension `.ini` files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, located in your `<Silk Test Classic installation directory>\extend` folder are copied to the `extend` directory of your project, regardless of what extension you have enabled. Do not rename the `extend` directory; this directory must exist in order for Silk Test Classic to open your project.

You can have Silk Test Classic automatically enable the appropriate extension using the basic workflow bar, or you can manually enable extensions. The current project uses the extension options in the extension `.ini` file copied to the `extend` directory of your project. Any modifications you make to the options for this enabled extension will be saved to the copy stored within the current project in the `extend` directory.

The `extend` directory is used only for local testing on the host machine. If you want to test on remote agent machines, you must copy the `.ini` files from the `extend` directory of your project to the `extend` directory on the target machines.

File references

Whether you are emailing, packaging, or adding files to a project, it is important to understand how Silk Test Classic stores the path of the file. The `.vtp` files of Silk Test Classic use relative paths for files on the same root drive and absolute paths for files with different root drives. The use of relative and absolute file paths is not configurable and cannot be overridden. If you modify the `.vtp` file to change file references from relative paths to absolute paths, the next time you open and close the project it will have relative paths and your changes will be lost.

Accessing Files Within Your Project

Working with Silk Test Classic projects makes it easy to access your files - once you have added a file to your project, you can open it by double-clicking it in the **Project Explorer**. The **Project Explorer** contains the following two tabs:

Tab	Description
-----	-------------

Files	Lists all of the files included in the project. From the Files tab, you can view, edit, add, and remove files from the project, as well as right-click to access menu options for each of the file types. From the Files tab, you can also add, rename, remove and work with folders within each category.
--------------	--

Global	Displays all the resources that are defined at a global level within the project's files. For example test cases, functions, classes, window declarations, and others. When you double-click an object on the Global tab, the file in which the object is defined opens and your cursor displays at the beginning of the line in which the object is defined. You can run and debug test cases and
---------------	---

Tab	Description
-----	-------------

	application states from the Global tab. You can also sort the elements that display within the folders on the Global tab.
--	---

Existing test sets do not display in the **Project Explorer** by default; you must convert them into projects.

Sharing a Project Among a Group

Apply the following guidelines to share a Silk Test Classic project among a group:

- Create the project in the location from which it will be shared. For example, you can create the project on a network drive.
- Ensure that testers create the same directory structure on their machines.

Project Explorer

Use the **Project Explorer** to view and work with all the resources within a Silk Test Classic project. You can access the **Project Explorer** by clicking:

- **File > Open Project** and specifying the project you want to open.
- **File > New Project** and creating a new project.
- **Project > View Explorer**, if you currently have a project open and do not have the **Project Explorer** view on.
- **Project > New Project** or **Open Project** on the **Basic Workflow** bar.

The resources associated with the project are grouped into categories. You can easily navigate among and access all of these resources using the **Files** and **Global** tabs. When you double-click a file on the **Files** tab, or an object on the **Global** tab, the file opens in the right pane. You can drag the divider to adjust the size of the **Project Explorer** windows and click **Project > Align** to change the orientation of the tabs from left to right.

Files tab

The **Files** tab lists all of the files that have been added to the project. The file name displays first, followed by the path. If files exist on a network drive, they are referenced using Universal Naming Conventions (UNC). Files are grouped into the following categories:

Category	Description
Profile	Contains project-specific initialization files, such as the <code>projectname.ini</code> and option sets files, which means <code>.opt</code> files, that are associated with the project.
Script	Contains test scripts, which means <code>.t</code> and <code>.g.t</code> files, that are associated with the project.
Include/Frame	Contains include files, which means <code>.inc</code> files, and frame/object files that are associated with the project.
Plan	Contains test plans and suite files, which means <code>.pln</code> and <code>.s</code> files, that are associated with the project.
Results	Contains results, which means <code>.res</code> and <code>.rex</code> files, that are associated with the project.
Data	Contains data associated with the project, such as Microsoft Word documents, text files, bitmaps, and others. Double-click the file to open it in the appropriate application. You must open files that are not associated with application types in the Windows Registry using the File/Open dialog box.

From the **Files** tab, you can view, edit, add, remove and work with files within the project. For example, to add a file to the project, right-click the category name, for example **Script**, and then click **Add File**. After

you have added the file, you can right-click the file name to view options for working with the file, such as record test case and run test case. Silk Test Classic functionality has not changed - it is now accessible through a project.

You can work with the folders within the categories on the **Files** tab, by adding, renaming, moving, and deleting folders within each category.

Global tab

The **Global** tab lists resources that are defined at a global level within the entire project. The resource name displays first, followed by the file in which it is defined. Resources contained within the project's files are grouped into the following categories:

- Records
- Classes
- Enums
- Window Declarations
- Testcases
- Appstates
- Functions
- Constants

From the **Global** tab, you can go directly to the location in which a global object or resource is defined. Double-click any object within the folders to go to the location in which the object is defined. Silk Test Classic opens the file and positions your cursor at the beginning of the line in which the object is defined.

You can also run and debug test cases and application states by right-clicking a test case or application state, and then selecting the appropriate option. For example, right-click a test case within the `Testcase` folder and then click **Run**. Silk Test Classic opens the file containing the test case you selected, and displays the **Run Testcase** dialog box with the selected test case highlighted. You can input argument values and run or debug the test case.

On the **Global** tab, you can sort the resources within each node by resource name, file name, or file date.



Note: Methods and properties are not listed on the **Global** tab since they are specific to classes or window declarations. You can access methods and properties by double-clicking the class or window declaration in which they are defined.

You cannot move files within the **Project Explorer**. For example, you cannot drag a script file under the **Frame** file node. However, you can drag the file to another folder within the same category node.



Note: If you change the location or name of a file included in your project, outside of Silk Test Classic, you must make sure the `projectname.vtp` contains the correct reference.

Creating a New Project

You can create a new project and add the appropriate files to the project, or you can have Silk Test Classic automatically create a new project from an existing file.

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by opening Silk Test Classic and clicking **Options > Save New Options Set**. You can add the options set to your project.

To create a new project:

1. In Silk Test Classic, click **File > New Project**, or click **Open Project > New Project** on the basic workflow bar. the **Create Project** dialog box appears.
2. Type a unique name for the project into the **Project Name** field.

If the project name already exists, you are prompted to specify a different project name or overwrite the existing project.

Project files with a `.vtp` (Verify Test Project) extension, `projectname.vtp`, and a `projectname.ini` are created in the directory that you specify in the **Save in** text box.

3. *Optional:* Type a description for the project into the **Description** field.

After you create the project, you can view the project description by right-clicking the `projectname` folder on the **Files** or **Global** tabs or clicking **Project > Project Description**.

4. Type the location at which you want to save the project into the **Save in** field.

You can also click **Browse** and specify the folder in which you want to save your project. The default location is `C:\Users\\Documents\Silk Test Classic Projects`.

5. Click **OK**. Silk Test Classic creates a `<Project name>` folder within the specified directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension `.ini` files, which are `appexpx.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project. Silk Test Classic then creates your project and displays nodes on the **Files** and **Global** tabs for the files and resources associated with this project.

6. Perform one of the following steps:

- If your test uses the Open Agent, configure the application to set up the test environment.
- If your test uses the Classic Agent, enable the appropriate extensions to test your application.

Opening an Existing Project

You can open a Silk Test Classic project as well as open an archived Silk Test Classic project. You can also open a Silk Test Classic project or archived project through the command line.

To open an existing project:

1. Click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar.

If you already have a project open, a dialog box opens informing you that the open project will be closed. If you associated Silk Test Classic file types with Silk Test Classic during installation, then you can open a Silk Test Classic project or package by double-clicking the `.vtp` or `.stp` file.

2. If you are opening a packaged Silk Test Classic project, which means an `.stp` file, you must perform the following steps:

- a) Indicate into what directory you want to unpack the project in the **Base path** text box. The files are unpacked to the directory you indicate in the **Base path** text box.
- b) Enter a password into the **Password** text box if the archived Silk Test Classic project was saved with a password.

If you open a package by double-clicking the `.stp` file, the base path is the directory that contains the `.stp` file.

When you select a location for unpacking the archive on the **Open Project** dialog box, Silk Test Classic uses that directory path, the base path, to substitute for the drive and root directory in the Use Path and Use Files paths.

The **Base path** and **Password** text boxes are enabled only if you are opening an `.stp` file.

3. On the **Open Project** dialog box, specify the project that you want to open, and then click **Open**.

If you open a project file (`.vtp`) by clicking **File > Open** command, the `projectname.vtp` file will open in the **4Test Editor**, but the project and its associated settings will not be loaded. Projects do not display in the recently opened files list. To close all open files within a project, click **Window > Close All**.

Converting Existing Tests to a Project

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by clicking **Options > Save New Options Set**. You can include the options set when you create your project.

To convert existing test sets to a project:

1. Create a new project.
2. Manually add the files to the project.

Using Option Sets in Your Project

To use an options set within your project, you must make sure that the options set is loaded into memory. You can tell if an options set is loaded by looking at the Silk Test Classic title bar. If `filename.opt` displays in the title bar, then the options set `filename.opt` is loaded. If an options set is loaded, it overrides the settings contained in the `projectname.ini` file.



Note: When an options set is loaded, the context menu options are available only for the loaded options set; these menu options are grayed out for `.ini` and `.opt` files that are not loaded.

You can load an options set into your project using any of the following methods:

- If the options set is included in your project, within the **Profile** node on the **Files** tab, right-click the options set that you want to load and then click **Open Options Set**.
- Right-click **Save New Options Set** to load the options set and add it under the **Profile** node on the **Files** tab.
- Use the **Options** menu; click **Options > Open Options Set**, browse to the options set (`.opt`) that you want to load, and then click **Open**.
- Load the options set at runtime using the `optionset` keyword. This loads the options set at the point in the plan file in which the options set is called. All test cases that follow use this options set.

If an options set was loaded when you closed Silk Test Classic, Silk Test Classic automatically re-loads this options set when you re-start Silk Test Classic.

To include an options set in your project, you can add the options set by right-clicking **Profile** on the **Files** tab, clicking **Add File**, selecting the options set you want to add to the project, and then clicking **OK**. You can also click **Save New Options Set**; this loads the options set and adds it under the **Profile** node on the **Files** tab.

Editing an Options Set

To edit an options set in your project:

1. On the **Files** tab, expand the **Profile** node.
2. Right-click the options set that you want to edit and click **Open Options Set**. The options set is loaded into memory.
3. Right-click the options set that you want to edit again and select the type of option you want to edit. For example Runtime, Agent, Extensions, and others.
4. Modify your options and then click **OK**. Your current settings are changed and saved to the `.opt` file.

If you want to change settings for future use, double-click the options set that you want to edit on the **Files** tab. This opens the options file in the Editor without loading the options file into memory. Changes you

make to the options set in the Editor will be saved, but will not take effect until you load the options set by selecting **Open Options Set** from the **Options** menu or the right-click shortcut.

Silk Test Classic File Types

Silk Test Classic uses the following types of files in the automated testing process, each with a specific function. The files marked with an * are required by Silk Test Classic to create and run test cases.

File Type	Extension	Description
Project	.vtp	Silk Test Classic projects organize all the resources associated with a test set and present them visually in the Project Explorer , making it easy to see, manage, and work within your test environment. The project file has a Verify Test Project (.vtp) extension and is organized as an .ini file; it stores the names and locations of files used by the project. Each project file also has an associated project initialization file: <code>projectname.ini</code> .
Exported Project	.stp	A Silk Test Project (.stp) file is a compressed file that includes all the data that Silk Test Classic exports for a project. A file of this type is created when you click File > Export Project . The .stp file includes the configuration files that are necessary for Silk Test Classic to set up the proper testing environment.
Testplan	.pln	An automated test plan is an outline that organizes and enhances the testing process, references test cases, and allows execution of test cases according to the test plan detail. It can be of type masterplan or of subplan that is referenced by a masterplan.
Test Frame*	.inc	A specific kind of include file that upon creation automatically captures a declaration of the AUT's main window including the URL of the Web application or path and executable name for client/server applications; acts as a central repository of information about the AUT; can also include declarations for other windows, as well as application states, variables, and constants.
4Test Script*	.t	Contains recorded and hand-written automated test cases, written in the 4Test language, that verify the behavior of the AUT.
Data-Driven Script	.g.t	Contains data-driven test cases that pull their data from databases.
4Test Include File	.inc	A file that contains window declarations, constants, variables, classes, and user defined functions.
Suite	.s	Allows sequential execution of several test scripts.
Text File	.txt	An ASCII file that can be used for the following: <ul style="list-style-type: none"> • Store data that will be used to drive a data driven test case. • Print a file in another document (Word) or presentation (PowerPoint). • Accompany your automation as a readme file. • Transform a tab-delimited plan into a Silk Test Classic plan.
Results File	.res	Is automatically created to store a history of results for a test plan or script execution.
Results Export File	.rex	A single compressed results file that you can relocate to a different machine. Click Results > Export to create a .rex file out of the existing results files of a project.
TrueLog File	.xlg	A file that contains the captured bitmaps and the logging information that is captured when TrueLog is enabled during a test case run.
Keyword-Driven Test File	.kdt	An executable file which contains keywords. Keyword-driven test files are used when testing with the keyword-driven testing methodology.

File Type	Extension	Description
Keyword Sequence	.kseq	A keyword-sequence file contains a combination of keywords, which are always executed in the same order. A keyword-sequence file cannot be executed on its own, only when included in a keyword-driven test. In context of a keyword-driven test, a keyword-sequence can be used as any other keyword. Keyword sequence files are used when testing with the keyword-driven testing methodology.

Organizing Projects

This section includes the topics that are available for organizing projects.

Adding Existing Files to a Project

You can add existing files to a project or create new files to add to the project. We recommend adding all referenced files to your project so that you can easily see and access the files, and the objects contained within them. Referenced files do not have to be included in the project. Plans and scripts will continue to run, provided the paths that are referenced are accurate.

When you add a file to a project, project files (.vtp files) use relative paths for files on the same root drive and absolute paths for files with different root drives. The use of relative and absolute files is not configurable and cannot be overridden.

To add an existing file to a project:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar, select the project to which you want to add a file, and then click **Open**.
2. On the **Project Explorer**, select the **Files** tab, right-click the node associated with the type of file you want to add, and then click **Add File**.

For example, to add a script file to the project, right-click **Script**, and then click **Add File**.

3. On the **Add File to Project** dialog box, specify the file you want to add to the open project, and then click **Open**.

The file name, followed by the path, displays under the appropriate category on the **Files** tab sorted alphabetically by name and is associated with the project through the `projectname.vtp` file. If files exist on a network drive, they are referenced using Universal Naming Conventions (UNC).

You can also add existing files to the project by clicking **Project > Add File**. Silk Test Classic automatically places the file in the appropriate node, based on the file type; for example if you add a file with a .pln extension, it will display under the **Plan** node on the **Files** tab. We do not recommend adding application .ini files or Silk Test Classic .ini files, which are `qaplans.ini`, `propset.ini`, and the `extension.ini` files, to your project. If you add object files, which are .to and .ino files, to your project, the files will display under the **Data** node on the **Files** tab. Objects defined in object files will not display in the **Global** tab. You cannot modify object files within the Silk Test Classic editor because object files are binary. To modify an object file, open the source file, which is a .t or .inc file, edit it, and then recompile.

Renaming Your Project

The `projectname.ini` and the `projectname.vtp` refer to each other; make sure the references are correct in both files when you rename your project.

To rename your project:

1. Make sure the project you want to rename is closed.

2. In Windows Explorer, locate the `projectname.vtp` and `projectname.ini` associated with the project name you want to change.
3. Change the names of `projectname.vtp` and `projectname.ini`. Make sure that you use the same `projectname` for both files.
4. In a text editor outside of Silk Test Classic, open `projectname.vtp`, change the reference to the `projectname.ini` file to the new name, and then save and close the file. Do not open the project in Silk Test Classic yet.
5. In a text editor outside of Silk Test Classic, open `projectname.ini`, change the reference to the `projectname.vtp` file to the new name, and then save and close the file.
6. In Silk Test Classic, open the project by clicking **File > Open Project** or **Open Project** on the basic workflow bar. The new project name displays.

Working with Folders in a Project

In addition to working with files, you can also add your own folders to all nodes listed on the **File** tab of the **Project Explorer**. For example, the **Files** tab of the **Project Explorer** can include notes.

You can also right-click a folder and click the following:

- **Expand All** to display all contents of a folder.
- **Collapse All** to collapse the contents of the folder.
- **Display Full Path** to show the full path for the contents.
- **Display Date/Time** to show creation information for the content file.

Adding a Folder to the Files Tab of the Project Explorer

You may add a folder to any of the categories (nodes) on the **Files** tab of the **Project Explorer**. You may not add a folder to the root project folder, nor change the titles of the root nodes.

To add a folder to a project:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab, right-click a folder and select **Add Folder**.
3. On the **Add Folder** dialog box, enter the name of the new folder, then click **OK**.

When you are naming a folder, you may use alphanumeric characters, underscore character, character space, or hyphens. Folder names may be a maximum of 256 characters long. Creating folders with more than 256 characters is possible, but Silk Test Classic will truncate the name when you save the project. The concatenated length of the names of all folders within a project may not exceed 256 characters. You may not use periods or parentheses in folder names. Within a node, folder names must be unique.

Moving Files and Folders

You may move an individual file or files between folders within the same category on the **Files** tab of the **Project Explorer**. You cannot move the predefined Silk Test Classic folders (nodes) such as Profile Script, Plan, Frame, and Data.

You may also move sub-folders within the same category on the **Files** tab. You cannot move sub-folders across categories.

To move a folder or file:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab. Click a file, a folder, or shift-click to select several files or folders, then drag the items to the new location.

3. Release the mouse to move the items.

There is no undo.

Removing a Folder from the Files tab of the Project Explorer

You may delete folders on the **Files** tab of the **Project Explorer**, however, you may not delete any of the predefined Silk Test Classic categories (nodes) such as Profile Script, Plan, Frame, and Data.



Note: There is no undo.

To remove a folder:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab, right-click a folder and select **Remove Folder** to delete it from the **Project Explorer**. If you select a folder with child folders or a folder that contains items, Silk Test Classic displays a warning before deleting the folder.

Renaming a Folder on the Files Tab of the Project Explorer

You may rename any folder that you have added to a project. You may not rename any of the predefined Silk Test Classic folders (nodes) such as Profile, Script, Include/Frame, Plan, Results, or Data.

To rename a folder:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar. Select a project, then click **Open**.
2. In the **Project Explorer**, click the **Files** tab, then navigate to the folder you want to rename.
3. Right-click the folder and select **Rename Folder**.
4. On the **Rename Folder** dialog box, enter the new name of the folder then click **OK**.

When naming a folder, you may use alphanumeric characters, underscore character, character space, or hyphens. Folder names may be a maximum of 64 characters long. You may not use periods or parentheses in folder names. Within a node, folder names must be unique.

Sorting Resources within the Global Tab of the Project Explorer

On the **Global** tab of the **Project Explorer**, you can sort the resources within each category (node) by resource name, file name, or file date.

To sort resources:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar, select the project whose elements you want to sort, and then click **Open**.
2. On the **Project Explorer**, click the **Global** tab, right-click the node associated with the type of element you want to sort, and then click **Sort by FileName** or **Sort by FileDate**.
The default is sort by element name.
3. Click **Ascending** or **Descending** to indicate how you want to organize the sort.
For example, to sort the elements of a script file by file date in reverse chronological order, right-click the **Script** node and select **Sort by FileDate**, then click **Descending**.

When you release the mouse, the elements are sorted by the parameters you selected.

Moving Files Between Projects

We recommend that you use **Export Project** to move projects, but if you want to move only a few files rather than an entire project, you can open the project in Silk Test Classic and remove the files that you want to move from the project. Move the files to their new location in Windows Explorer, and then add the files back to the currently open project.

You can also move your project by opening the `projectname.vtp` and `projectname.ini` files in a text editor outside of Silk Test Classic and updating references to the location of source files. However, we recommend that you have strong knowledge of your files and how the `partner` and `projectname.ini` files work before attempting this. We advise you to use great caution if you decide to edit the `projectname.vtp` and `projectname.ini` files.

Removing Files from a Project

You cannot remove the `projectname.ini` file.

To remove a file from a project:

1. Click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar.
2. Click the plus sign [+] to expand the node associated with the type of file you want to remove, and then choose one of the following:
 - Right-click the file you want to remove, and then click **Remove File**.
 - Select the file in the **Project Explorer** and press the **Delete** key.
 - Select the file you want to remove on the **Files** tab, and then click **Project > Remove File**.

The file is removed from the project and references to the file are deleted from the `projectname.vtp`. The file itself is not deleted; it is just removed from the project.

Turning the Project Explorer View On and Off

The **Project Explorer** view is the default. If you do not want to view the **Project Explorer**, uncheck **Project > View Explorer**. You can continue to work with your files within the project, you just will not see the **Project Explorer**.

To turn **Project Explorer** view on, check **Project > View Explorer**.

If you do not want to use projects in Silk Test Classic, close the open project, if any, by clicking **File > Close Project**, and then use Silk Test Classic as you would have in the past.

Viewing Resources Within a Project

1. Click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar and select the project that you want to open.
2. Click one of the following:
 - The **Files** tab to view all the files associated with the open project.
 - The **Global** tab to view global objects defined in the files associated with the project.
3. To close all open files within a project, click **Window > Close All**.

Packaging a Silk Test Classic Project

You can package your Silk Test Classic project into a single compressed file that you can relocate to a different computer. When you unpack your project you will have a fully functional set of test files. This is useful if you need to relocate a project, email a project to a co-worker, or send a project to technical support.

Source files included in the packaged project

When you package a project, Silk Test Classic includes all of the source files, meaning the related files used by a project, such as:

Description	Extension
plan files	.pln
script files	.t
include files	.inc
suite files	.s
results files (optional)	.res and .rex
data files	-

Silk Test Classic takes these files and bundles them up into a new file with an .stp extension. The .stp file includes the configuration files necessary for Silk Test Classic to set up the proper testing environment such as `project.ini`, `testplan.ini`, `optionset.opt` files, and any .ini files found in the `...\Silk Test Classic projects\<Project name>\extend` directory.

You have the option of including .res and .rex files when you package a Silk Test Classic project because these files are sometimes quite large and not necessary to run the project.

Relative paths in comparison to absolute paths

When you work with Silk Test Classic projects, the files that make up the project are identified by pathnames that are either absolute or relative. A relative pathname begins at a current folder or some number of folders up the hierarchy and specifies the file's location from there. An absolute pathname begins at the root of the file system (the topmost folder) and fully specifies the file's location from there. For example:

Absolute path `C:\Users\<Current user>\Documents\Silk Test Classic Projects\<Project name>\options.ini`

Relative path `..\tesla\Silk Test\options\options.ini` or `SUSDir\options.inc`

When you package a project, Silk Test Classic checks to make sure that the paths used within the project are properly maintained. If you try to compress a project containing ambiguous paths, Silk Test Classic displays a warning message. Silk Test Classic tracks the paths in a project in a log file.

Including all files needed to run tests

Files associated with a project, but not necessary to run tests, for example bitmap or document files, which you have manually added to the project are included when Silk Test Classic packages a project.

If Silk Test Classic finds any `include:`, `script:`, or `use:` statements in the project files that refer to files with absolute paths, `c:\program files\Silk\Silk Test\`, Silk Test Classic verifies if you have checked the **Use links for absolute files?** check box on the **Export Project** or on the **Email Project** dialog boxes.

- If you check the **Use links for absolute files?** check box, Silk Test Classic treats any file referenced by an absolute path in an `include`, `script`, or `use` statement as a placeholder and does not include those files in the package. For example, if there are use files within the **Runtime Options** dialog box referred to as `"q:\qaplans\SilkTest\frame.inc"` or `"c"`, these files are not included in the package. The assumption is that these files will also be available from wherever you unpack the project.
- If you uncheck the **Use links for absolute files?** check box, Silk Test Classic includes the files referenced by absolute paths in the packaged project. For example, if the original file is stored on `c:\temp\myfile.t`, when unpacked at the new location, the file is placed on `c:\temp\myfile.t`.

The following table compares the results of packaging projects based on whether there are any absolute file references in your source files, and how you respond to the **Use links for absolute files?** check box on the **Export Project** or on the **Email Project** dialog boxes.

Any absolute references in source files?	Use links for absolute files?	Results
No	Checked or unchecked	Package unpacks to any location.
Yes	Checked	Files referenced by absolute paths are not included in the packaged project.
Yes	Unchecked	Files referenced by absolute paths are put into a ZIP file within the packaged project.



Note:

- If there are any source files located on a different drive than the .vtp project file, and if there are files referenced by absolute paths in the source files, Silk Test Classic treats the source files as referenced by absolute paths. The assumption is that the absolute paths will be available from the new location. Silk Test Classic therefore puts the files into a zip file within the packaged project for you to unpack after you unpack the project.
- Files not included in the package - The assumption is that since these files are referenced by absolute paths, these same files and paths will be available when the files are unpacked. On unpacking, Silk Test Classic warns you about these files and lists them in a log file (manifest.XXX).
- ip files – Because you elected not to use links for files referenced by absolute paths, these files are put into a zip file within the packaged project. The zip file is named with the root of the absolute path. For example, if the files are located on c:/, the zip file is named c.zip.

Tips for successful packaging and unpacking

For best results when packaging and unpacking Silk Test Classic projects:

- Put your .vtp project file and source files on the same drive.
- Use relative paths to reference the following:
 - include statements
 - options sets
 - use paths set within the **Runtime Options** dialog box
 - use statements in 4Test scripts
 - script statements
- Uncheck the default **Use links for absolute files?** check box if your source files are on a different drive as the .vtp project file and if there are files referenced by absolute paths in your source files.

Packaging with Silk Test Classic Runtime and the Agent

If you are running Silk Test Classic Runtime, you may not package or email a project.

If you are running the Agent, you may package or email a project.

Emailing a Project

Emailing a project automatically packages a Silk Test Classic project and then emails it to an email address. In order for this to work, you must have an email client installed on the computer that is running Silk Test Classic.

You cannot email a project if you are running Silk Test Classic Runtime.

One of the options you can select before emailing is to compile your project. If a compile error occurs, Silk Test Classic displays a warning message, and you can opt to continue or to cancel the email.

Silk Test Classic supports any MAPI-compliant email clients such as Outlook Express.

The maximum size for the emailed project is determined by your email client. Silk Test Classic does not place any limits on the size of the project.

To email your project:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar. Select a project, then click **Open**.

2. Click **File > Email Project**.

You can only email a project if you have that project open.

3. On the **Email Project** dialog box, type the email address where you want to send the Silk Test Classic project.

For example, enter `support@acme.com` to send a package to Acme Technical Support.



Tip: If you are not sure about the email address, you can just enter some text here. Once you click **OK**, this information is passed to your default mail system where you can correct the address.

4. *Optional:* Check the **Create file references for files with absolute paths** check box to use links for any absolute file reference.

By default, this check box is checked.



Note: Using references for absolute paths produces a smaller package that can be opened more quickly.

5. *Optional:* Check the **Compile before exporting** check box to compile the project before emailing it.

By default, this check box is checked.

6. *Optional:* Check the **Include results files of all tests (.res, .rex, and .xlg)** check box to include `.res`, `.rex`, and `.xlg` results files in the emailed project.

Only results files that were added to the **Results** folder of the project are emailed. By default, this check box is unchecked.

7. *Optional:* Check the **Include extend.ini file** check box to include extensions that you have configured for target machines in the project that you export.

Typically, when working with a project, extension configurations are stored in the project specific `project.ini` file. However, you can also configure extensions for target machines in the `extend.ini` file, by using the **Extension Enabler**, a tool which is available from the **Start** menu. By default, this check box is unchecked.

8. *Optional:* Check the **Protect Silk Test Classic package files with password** check box to secure the compressed file with a password.

If you have secured the compressed file with a password, you cannot extract any files from the compressed package without specifying this password. This option is available only when you have checked the **Export to single Silk Test Classic package** check box.

a) Type the password into the **Enter password** field.

A password may include up to 79 alphanumeric characters.

b) Re-enter the password into the **Confirm password** field to confirm it.

9. Click **OK**. If you opted to compile the project before packaging it, Silk Test Classic displays a warning message if any file failed to compile. Silk Test Classic opens a new email message and attaches the packaged project to a message. You can edit the recipient, add a subject line, and text, just as you can for any outgoing message.

10. Click **Send** to add the project to your outgoing queue. If your email client is already open, your message is sent automatically. If your email client was not open, the message is placed in your outgoing queue.



Note: If the email process does not finish successfully, Micro Focus recommends deleting any partially packaged project or draft email message and restarting the email process.

Exporting a Project

Exporting a Silk Test Classic project lets you copy all the files associated with a project to a directory or a single compressed file in a directory.

You cannot export a project if you are running Silk Test Classic Runtime.

Silk Test Classic will not change the file creation dates when copying the project's files.

One of the options you can select before exporting is to compile your project. If a compile error occurs, Silk Test Classic displays a warning message, and you can opt to continue or to cancel the compile.

To export your project:

1. If your project is not already open, click **File > Open Project** or click **Open Project > Open Project** on the basic workflow bar. Select a project, then click **Open**.

2. Click **File > Export Project**.

You can only export a project if you have the project open.

3. On the **Export Project** dialog box, enter the directory to which you want to export the project or click



to locate the export folder.

The default location is the parent directory of the project folder, which means the folder containing the project file, not the project's current location.

4. Check the **Export to single Silk Test Classic package** check box if you want to package the Silk Test Classic project into a single compressed file.

5. *Optional:* Check the **Create file references for files with absolute paths** check box to use links for any absolute file reference.

By default, this check box is checked.



Note: Using references for absolute paths produces a smaller package that can be opened more quickly.

6. *Optional:* Check the **Compile before exporting** check box to compile the project before exporting it.

By default, this check box is checked.

7. *Optional:* Check the **Include results files of all tests** check box to include `.res` and `.rex` results files in the exported project or package.

Only results files that were added to the **Results** folder of the project are exported. By default, this check box is unchecked.

8. *Optional:* Check the **Include extend.ini file** check box to include extensions that you have configured for target machines in the project that you export.

Typically, when working with a project, extension configurations are stored in the project specific `project.ini` file. However, you can also configure extensions for target machines in the `extend.ini` file, by using the **Extension Enabler**, a tool which is available from the **Start** menu. By default, this check box is unchecked.

9. *Optional:* Check the **Protect Silk Test Classic package files with password** check box to secure the compressed file with a password.

If you have secured the compressed file with a password, you cannot extract any files from the compressed package without specifying this password. This option is available only when you have checked the **Export to single Silk Test Classic package** check box.

- a) Type the password into the **Enter password** field.

A password may include up to 79 alphanumeric characters.

- b) Re-enter the password into the **Confirm password** field to confirm it.

10. Click **OK**. Silk Test Classic determines all the files necessary for the project and copies them to the selected directory or compresses them into a package. Silk Test Classic displays a warning message if any of the files could not be successfully packaged and gives you the option of continuing.



Tip: If the export process does not finish successfully, Micro Focus recommends deleting any partially packaged project and restarting the export process.

Troubleshooting Projects

This section provides solutions to common problems that you might encounter when you are working with projects in Silk Test Classic.

Files Not Found When Opening Project

If, when opening your project, Silk Test Classic cannot find a file in the location referenced in the project file, which is a `.vtp` file, an error message displays noting the file that cannot be found.

Silk Test Classic may not be able to find files that have been moved or renamed outside of Silk Test Classic, for example in Windows Explorer, or files that are located on a shared network folder that is no longer accessible.

- If Silk Test Classic cannot find a file in your project, we suggest that you note the name of missing file, and click **OK**. Silk Test Classic will open the project and remove the file that it cannot find from the project list. You can then add the missing file to your project.
- If Silk Test Classic cannot open multiple files in your project, we suggest you click **Cancel** and determine why the files cannot be found. For example a directory might have been moved. Depending upon the problem, you can determine how to make the files accessible to the project. You may need to add the files from their new location.

Silk Test Classic Cannot Load My Project File

If Silk Test Classic cannot load your project file, the contents of your `.vtp` file might have changed or your `.ini` file might have been moved.

If you remove or incorrectly edit the `ProjectIni=` line in the `ProjectProfile` section of your `<projectname>.vtp` file, or if you have moved your `<projectname>.ini` file and the `ProjectIni=` line no longer points to the correct location of the `.ini` file, Silk Test Classic is not able to load your project.

To avoid this, make sure that the `ProjectProfile` section exists in your `.vtp` file and that the section refers to the correct name and location of your `.ini` file. Additionally, the `<projectname>.ini` file and the `<projectname>.vtp` file refer to each other, so ensure that these references are correct in both files. Perform these changes in a text editor outside of Silk Test Classic.

Example

The following code sample shows a sample `ProjectProfile` section in a `<projectname>.vtp` file:

```
[ProjectProfile]
ProjectIni=C:\Program Files\<Silk Test install directory>
\SilkTest\Projects\<projectname>.ini
```

Silk Test Classic Cannot Save Files to My Project

You cannot add or remove files from a read-only project. If you attempt to make any changes to a read-only project, a message box displays indicating that your changes will not be saved to the project.

For example, Unable to save changes to the current project. The project file has read-only attributes.

When you click **OK** on the error message box, Silk Test Classic adds or removes the file from the project temporarily for that session, but when you close the project, the message box displays again. When you re-open the project, you will see your files have not been added or removed.

Additionally, if you are using Microsoft Windows 7 or later, you might need to run Silk Test Classic as an administrator. To run Silk Test Classic as an administrator, right-click the Silk Test Classic icon in the **Start Menu** and click **Run as administrator**.

Silk Test Classic Does Not Run

The following table describes what you can do if Silk Test Classic does not start.

If Silk Test Classic does not run because it is looking for the following:	You can do the following:
Project files that are moved or corrupted.	Open the <code>SilkTestClassic.ini</code> file in a text editor and remove the <code>CurrentProject=</code> line from the <code>ProjectState</code> section. Silk Test Classic should then start, however your project will not open. You can examine your <code><projectname>.ini</code> and <code><projectname>.vtp</code> files to determine and correct the problem. The following code example shows the <code>ProjectState</code> section in a sample <code>partner.ini</code> file: <pre>[ProjectState] CurrentProject=C:\Program Files \SilkTest\install\directory> \SilkTest\Examples\ProjectName.vtp</pre>
A <code>testplan.ini</code> file that is corrupted.	Delete or rename the corrupted <code>testplan.ini</code> file, and then restart Silk Test Classic.

My Files No Longer Display In the Recent Files List

After you open or create a project, files that you had recently opened outside of the project do no longer display in the **Recent Files** list.

Cannot Find Items In Classic 4Test

If you are working with Classic 4Test, objects display in the correct nodes on the **Global** tab, however when you double-click an object, the file opens and the cursor displays at the top of the file, instead of in the line in which the object is defined.

Editing the Project Files

You require good knowledge of your files and how the `partner` and `<projectname>.ini` files work before attempting to edit these files. Be cautious when editing the `<projectname>.vtp` and `<projectname>.ini` files.

To edit the `<projectname>.vtp` and `<projectname>.ini` files:

1. Update the references to the source location of your files. If the location of your `projectname.vtp` and `projectname.ini` files has changed, make sure you update that as well. Each file refers to the other.

The `ProjectProfile` section in the `projectname.vtp` file is required. Silk Test Classic will not be able to load your project if this section does not exist.

1. Ensure that your project is closed and that all the files referenced by the project exist.
2. Open the `<projectname>.vtp` and `<projectname>.ini` files in a text editor outside of Silk Test Classic.



Note: Do not edit the `projectname.vtp` and `projectname.ini` files in the 4Test Editor.

3. Update the references to the source location of your files.
4. The `<projectname>.vtp` and `<projectname>.ini` files refer to each other. If the relative location of these files has changed, update the location in the files.

The `ProjectProfile` section in the `<projectname>.vtp` file is required. Silk Test Classic is not able to load your project if this section does not exist.

Project Description Dialog Box

Use the **Project Description** dialog box to view or edit the description of the current project.

Open a project and click **Project > Project Description**.

Description Displays the description of this project. The original description is the text that you typed in the **Description** field on the **Create Project** dialog box or the **AutoGenerate Project** dialog box when you created the project. You can modify the project description as necessary, typing up to 1024 characters. The project description is stored in the **System Settings** section of the `projectname.vtp` file.

Enabling Extensions for Applications Under Test

This functionality is supported only if you are using the Classic Agent.

This section describes how you can use extensions to extend the capabilities of a program or the data that is available to the program.

An extension is a file that serves to extend the capabilities of, or the data available to, a basic program. Silk Test Classic provides extensions for testing applications that use non-standard controls in specific development and browser environments.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Related Files

If you are using a project, the extension configuration information is stored in the `partner.ini` file. If you are not using a project, the extension configuration information is stored in the `extend.ini` file.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files** location in the **Runtime Options** dialog box. Extensions that use technologies on the Classic Agent are located in the `<Silk Test Classic project directory>\extend\` directory.

Extensions that Silk Test Classic can Automatically Configure

This functionality is supported only if you are using the Classic Agent.

Using the **Basic Workflow**, Silk Test Classic can automatically configure extensions for many development environments, including:

- Browser applications and applets running in one of the supported browsers.
- .NET standalone Windows Forms applications.
- Standalone Java and Java AWT applications.
- Java Web Start applications and InstallAnywhere applications and applets.
- Java SWT applications.
- Visual Basic applications.
- Client/Server applications.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

You cannot enable extensions for Silk Test Classic (`partner.exe`), Classic Agent (`agent.exe`), or Open Agent (`openAgent.exe`).

You can also click **Tools > Enable Extensions** to have Silk Test Classic automatically set your extension.

If the **Basic workflow** does not support your configuration, you can enable the extension manually.

If you use the Classic Agent, the **Basic Workflow** does not automatically configure browser applications containing ActiveX objects. To configure a browser application with ActiveX objects, check the **ActiveX** check box in the row for the extension that you are enabling in the **Extensions** dialog box. Or use the Open Agent.

Extensions that Must be Set Manually

This functionality is supported only if you are using the Classic Agent.

Using the **Basic Workflow**, Silk Test Classic can automatically enable extensions for many different development environments. If the **Basic Workflow** does not support your configuration or you prefer to enable extensions manually, enable the extension on your host machine and enable the extension on your target machine, regardless of whether the application you plan to test will run locally or on remote machines. Enable extensions manually if you:

- Want to change your currently enabled extension.
- Want to enable additional options for the extension you are using, such as Accessibility, Active X, or Java.
- Are testing embedded browser applications using the Classic Agent, for example, if DOM controls are embedded within a Windows Forms application.
- Are testing an application that does not have a standard name.

If you are testing Web applications using the Classic Agent, Silk Test Classic enables the extension associated with the default browser you specified on the **Select Default Browser** dialog box during the Silk Test Classic installation. If you want to use the extension you specified during the Silk Test Classic installation, you do not need to complete this procedure unless you need additional options, such as Accessibility, Java, or ActiveX.

If you are not testing Java but do have Java installed, we recommend that you disable the classpath before using Silk Test Classic.

Silk Test Classic automatically enables Java support in the browser if your web page contains an applet. The **Enable Applet Support** check box on the **Extension Settings** dialog for browser is automatically selected when the **Enable Extensions** workflow detects an applet. You can uncheck the check box to prevent Silk Test Classic from loading the extension. If no applet is detected, the check box is not available.

Extensions on Host and Target Machines

This functionality is supported only if you are using the Classic Agent.

You must define which extensions Silk Test Classic should load for each application under test, regardless of whether the application will run locally or on remote machines. You do this by enabling extensions on your host machine and on each target machine before you record or run tests.

Extensions on the host machine

On the host machine, we recommend that you enable only those extensions required for testing the current application. Extensions for all other applications should be disabled on the host to conserve memory and other system resources. By default, the installation program:

- Enables the extension for your default Web browser environment on the host machine.
- Disables extensions on the host machine for all other browser environments.
- Disables extensions for all other development environments.

When you enable an extension on the host machine, Silk Test Classic does the following:

- Adds the include file of the extension to the **Use Files** text box in the **Runtime Options** dialog box, so that the classes of the extension are available to you.
- Makes sure that the classes defined in the extension display in the **Library Browser**. Silk Test Classic does this by adding the name of the extension's help file, which is `browser.ht`, to the **Help Files For**

Library Browser text box in **General Options** dialog box and recompiling the help file used by the **Library Browser**.

- Merges the property sets defined for the extension with the default property sets. The web-based property sets are in the `browser.ps` file in the `Extend` directory. The file defines the following property sets: Color, Font, Values, and Location.

Extensions on the target machine

The **Extension Enabler** dialog box is the utility that allows you to enable or disable extensions on your target machines. All information that you enter in the **Extension Enabler** is stored in the `extend.ini` file and allows the Agent to recognize the non-standard controls you want to test on target machines.

Enabling Extensions Automatically Using the Basic Workflow

An extension is a file that serves to extend the capabilities of, or data available to, a more basic program. Silk Test Classic provides extensions for testing applications that use non-standard controls in specific development and browser environments.

If you are testing a generic project that uses the Classic Agent, perform the following procedure to enable extensions:

1. Start the application or applet for which you want to enable extensions.
2. Start Silk Test Classic and make sure the basic workflow bar is visible. If it is not, click **Workflows > Basic** to enable it.
If you do not see **Enable Extensions** on the workflow bar, ensure that the default agent is set to the Classic Agent.
3. If you are using Silk Test Classic projects, click **Project** and open your project or create a new project.
4. Click **Enable Extensions**.
You cannot enable extensions for Silk Test Classic (`partner.exe`), the Classic Agent (`agent.exe`), or the Open Agent (`openAgent.exe`).
5. Select your test application from the list on the **Enable Extensions** dialog box, and then click **Select**.
6. If your test application does not display in the list, click **Refresh**. Or, you may need to add your application to this list in order to enable its extension.
7. Click **OK** on the **Extension Settings** dialog box, and then close and restart your application.
8. If you are testing an applet, the **Enable Applet Support** check box is checked by default.
9. When the **Test Extension Settings** dialog box opens, restart your application in the same way in which you opened it; for example, if you started your application by double-clicking the `.exe`, then restart it by double-clicking the `.exe`.
10. Make sure the application has finished loading, and then click **Test**. When the test is finished, a dialog box displays indicating that the extension has been successfully enabled and tested. You are now ready to begin testing your application or applet. If the test fails, review the troubleshooting topics.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box.

Enabling Extensions on a Host Machine Manually

This functionality is supported only if you are using the Classic Agent.

Using the **Basic workflow**, Silk Test Classic can automatically enable extensions for many different development environments. If you would rather enable the extension manually, or the basic workflow does not support your configuration, follow the steps described in this topic.

A host machine is the system that runs the Silk Test Classic software process, in which you develop, edit, compile, run, and debug 4Test scripts and test plans.

There is overhead to having more than one browser extension enabled, so you should enable only one browser extension unless you are actually testing more than one browser in an automated session.

1. Start Silk Test Classic and click **Options > Extensions**.
2. If you are testing a client/server project, rich internet application project, or a generic project that uses the Classic Agent, perform the following steps:
 - a) On the **Extensions** dialog box, click the extension you want to enable. You may need to add your application to this list in order to enable its extension.
 - b) Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate.
 - c) Disable other extensions that you will not be using by selecting **Disabled** in the **Primary Extension** field. To disable a Visual Basic extension, uncheck the **ActiveX** check box for the Visual Basic application.
 - d) Click **OK**.

Manually Enabling Extensions on a Target Machine

This functionality is supported only if you are using the Classic Agent.

Using the basic workflow, Silk Test Classic can automatically enable extensions for many different development environments. If you would rather enable the extension manually, or the basic workflow does not support your configuration, follow the steps described in this topic.

A target machine is a system (or systems) that runs the 4Test Agent, which is the software process that translates the commands in your scripts into GUI-specific commands, in essence, driving and monitoring your applications under test. One Agent process can run locally on the host machine, but in a networked environment, any number of Agents can run on remote machines.

If you are running local tests, that is, your target and host are the same machine, complete this procedure and enable extensions on a host machine manually.

1. Make sure that your browser is closed.
2. From the Silk Test Classic program group, choose **Extension Enabler**. To invoke the **Extension Enabler** on a remote non-Windows target machine, run `extinst.exe`, located in the directory on the target machine in which you installed the Classic Agent.
3. Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate. To get information about the files used by an extension, select an extension and click **Details**. You may need to add your application to this list in order to enable its extension.
4. Click **OK** to close the **Extension Enabler** dialog box.

If you enable support for ActiveX in this dialog box, make sure that it is enabled in the **Extensions** dialog box as well.

5. Restart your browser, if you enabled extensions for web testing.

Once you have set your extension(s) on your target and host machines, verify the extension settings to check your work. Be sure to consider how you want to work with borderless tables. If you are testing non-Web applications, you must disable browser extensions on your host machine. This is because the recovery system works differently when testing Web applications than when testing non-Web applications. For more information about the recovery system for testing Web applications, see Web applications and the recovery system. When you select one or both of the Internet Explorer extensions on the host machine's **Extension** dialog box, Silk Test Classic automatically picks the correct version of the host machine's Internet Explorer application in the **Runtime Options** dialog box. If the target

machine's version of Internet Explorer is not the same as the host machine's, you must remember to change the target machine's version.

Enabling Extensions for Embedded Browser Applications that Use the Classic Agent

This functionality is supported only if you are using the Classic Agent.

To test an embedded browser application, enable the Web browser as the primary extension for the application in both the **Extension Enabler** and in the Silk Test Classic **Extensions** dialog boxes. For instance, if you are testing an application with DOM controls that are embedded within a .NET application, follow the following instructions to enable extensions.

1. Click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Tools > Extension Enabler** or (in Microsoft Windows 10) **Start > Silk > Extension Enabler**.
2. Browse to the location of the application executable.
3. Select the executable file and then click **Open**.
4. Click **OK**.
5. From the **Primary Extension** list box, select the DOM extension for the application that you added.
6. Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate.
For example, to test a .NET application with embedded Web controls, select a browser in the **Primary Extension** list box and check the .NET check box for the application within the grid.
7. Click **OK**.
8. Start Silk Test Classic and then choose **Options > Extensions**. The **Extensions** dialog box opens.
9. Click **New**.
10. Browse to the location of the application executable.
11. Select the executable file and then click **Open**.
12. Click **OK**.
13. From the **Primary Extension** list box, select the DOM extension for the application that you added.
14. Enable other extensions, such as Java, ActiveX, Accessibility, and .NET, as appropriate.
For example, to test a .NET application with embedded Web controls, select a browser in the **Primary Extension** list box and check the .NET check box for the application within the grid.
15. Click **OK**.
16. Restart Silk Test Classic.





Note: The IE DOM extension may not detect changes to a web page that occur when JavaScript replaces a set of elements with another set of elements without changing the total number of elements. To force the DOM extension to detect changes in this situation, call the `FlushCache()` method on the top-level browserchild for the embedded browser. This problem might occur more often for embedded browsers than for browser pages, because Silk Test Classic is not notified of as many browser events for embedded browsers. Also call `FlushCache()` if you get a `Coordinate out of bounds` exception when calling a method, for example `Click()`, on an object that previously had been scrolled into view. The `BrowserPage` window identifier is not valid when using embedded browsers because the default browser type is `'(none)'` (NULL).

Enabling Extensions for HTML Applications (HTAs)

This functionality is supported only if you are using the Classic Agent.

You must enable extensions on the host and target machines manually in order to use HTML applications (HTAs).

Before you begin, create a project that uses the Classic Agent.

1. Click **Options > Extensions** to open the **Extensions** dialog box.
2. Click **New** to open the **Extension Application** dialog box.
3. Click  to navigate to the location of the .hta file that you want to enable. If the file name contains spaces, be sure to enclose the name in quotation marks.
4. Select the .hta file and then click **Open**.
5. Click **OK**.
6. In the **Primary Extension** column next to the .hta application that you just enabled, select **Internet Explorer**.
7. Click **OK**.
8. Click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Tools > Extension Enabler** or (in Microsoft Windows 10) **Start > Silk > Extension Enabler**. (Or use the command line to launch "C:\Program Files\Silk\SilkTest\Tools\extinst.exe".)
9. On the **Extension Enabler** dialog box, click **New** to open the **Extension Application** dialog box.
10. Click  to navigate to the location of the .hta file that you want to enable. If the file name contains spaces, be sure to enclose the name in quotation marks.
11. Select the .hta file and then click **Open**.
12. Click **OK**.
13. In the **Primary Extension** column next to the .hta application that you just enabled, select **Internet Explorer**.
14. Click **OK**.

Adding a Test Application to the Extension Dialog Boxes

This functionality is available only for projects or scripts that use the Classic Agent.

You must manually add the following applications to the **Extensions** dialog box and the **Extension Enabler** dialog box:

- Applications that are embedded in Web pages and use the Classic Agent.
- All test applications that do not have standard names and use the Classic Agent.
- When you add a test application to the **Extensions** dialog box on the host machine, you should immediately add it to the **Extension Enabler** dialog box on each target machine on which you intend to test the application.

You may also add new applications by duplicating existing applications and then changing the application name.

To add a test application to the **Extension** dialog boxes:

1. Click **Options > Extensions** to open the **Extensions** dialog box, or open the **Extension Enabler** dialog box from the Silk Test program group.
2. If you are testing a client/server project, Rich Internet Application project, or a generic project that uses the Classic Agent, perform the following steps:
 - a) Click **New** to open the **Extension Application** dialog box.
 - b) Click ... to browse to the application's executable or DLL file.

Separate multiple application names with commas. If the executable name contains spaces, be sure to enclose the name in quotation marks.

- c) Select the executable file and then click **Open**.
 - d) Click **OK**.
3. Click **OK** to close the dialog box.

Verifying Extension Settings

This functionality is available only for projects or scripts that use the Classic Agent.

If the extension settings for the host and target machines do not match, neither extension will load properly.

- To see the target machine setting, choose **Options > Extensions**. Verify that the Primary Extension is enabled and other extensions are enabled, if appropriate. If you enabled a browser extension, you can also verify the extension settings on the target machine by starting the browser and Silk Test Classic, and then right-clicking the task bar Agent icon and selecting **Extensions > Detail**.
- To verify that the setting on the host machine is correct, choose **Options > Runtime**. Make sure that the default browser in the **Default Browser** field on the **Runtime Options** dialog box is correct.

Why Applications do not have Standard Names

This functionality is supported only if you are using the Classic Agent.

In the following situations applications might not have standard names, in which case you must add them to the **Extension Enabler** dialog box and the **Extensions** dialog box:

- Visual Basic applications can have any name, and therefore the Silk Test Classic installation program cannot add them to the dialog box automatically.
- You are running an application developed in Java as a stand-alone application, outside of its normal runtime environment.
- You have explicitly changed the name of a Java application.

Duplicating the Settings of a Test Application in Another Test Application

This functionality is supported only if you are using the Classic Agent.

You can add new applications to the **Extension Enabler** dialog box or the **Extensions** dialog box by duplicating existing applications and renaming the new application. All the settings of the original application, that is, primary extension, other extensions, or options set on the **Extensions** dialog box, are copied.

You can only duplicate applications that you entered manually and that use the Classic Agent.

To copy a test application's settings into another application:

1. Click **Options > Extensions** to open the **Extensions** dialog box, or open the **Extension Enabler** dialog box from the Silk Test Classic program group.
2. Select the application that you want to copy.
3. Click **Duplicate**. The **Extension Application** dialog box opens.
4. Type the name of the new application you want to copy.
Separate multiple application names with commas.
5. Click **OK** to close the **Extension Application** dialog box. The new applications display in the dialog box you opened.
6. Click **OK** to close the dialog box.

Deleting an Application from the Extension Enabler or Extensions Dialog Box

This functionality is supported only if you are using the Classic Agent.

After completing your testing of an application or if you make a mistake, you might want to delete the application from the **Extension Enabler** dialog box or the **Extensions** dialog box. You can delete only applications that you have entered manually. Visual Basic applications fall into this category.

To remove an application from the **Extension Enabler** or **Extensions** dialog box:

1. Click **Options > Extensions** to open the **Extensions** dialog box, or open the **Extension Enabler** dialog box from the Silk Test Classic program group.
2. Select the application that you want to delete from the dialog box.
3. Click **Remove**. The application name is removed from the dialog box.
4. Click **OK**.

Disabling Browser Extensions

This functionality is supported only if you are using the Classic Agent.

1. In Silk Test Classic, choose **Options > Extensions**.
2. From the **Primary Extension** list, select **Disabled** for the extension you want to disable.
3. In the **Other extensions** field, uncheck any checked check boxes.
4. Click **OK**.

If you are testing non-Web applications, you must disable browser extensions on your host machine. This is because the recovery system works differently when testing Web applications than when testing non-Web applications.

Comparison of the Extensions Dialog Box and the Extension Enabler Dialog Box

This functionality is supported only if you are using the Classic Agent.

The **Extensions** dialog box and the **Extension Enabler** dialog box look similar; they are both based on a grid and have identical column headings and have some of the same buttons. However, they configure different aspects of the product:

	Extensions Dialog Box	Extension Enabler Dialog Box
Enables AUTs and extensions	On host machine	On target machines
Provides information for	Silk Test Classic	Agent
Available from	Options menu	Silk Test Classic program group
Information stored in	<code>partner.ini</code>	<code>extend.ini</code>
When to enable/disable AUTs and extensions	Enable the AUTs and extensions you want to test now; disable others.	Enable all AUTs and extensions you ever intend to test. No harm in leaving them enabled, even if you are not testing them now.

	Extensions Dialog Box	Extension Enabler Dialog Box
What you specify on each:	<ul style="list-style-type: none"> • Yes, according to the type 	<ul style="list-style-type: none"> • Yes, according to the type
<ul style="list-style-type: none"> • Primary environment • Java or ActiveX, if required • Accessibility 	<ul style="list-style-type: none"> • Enable and set options • Enable and set options 	<ul style="list-style-type: none"> • Enable only • Enable only
What installation does:	<ul style="list-style-type: none"> • Displayed and enabled 	<ul style="list-style-type: none"> • Displayed and enabled
<ul style="list-style-type: none"> • Default browser (If any) • Other browsers (if any) • Java runtime environment • Oracle Forms runtime environment • Visual Basic 5 & 6 	<ul style="list-style-type: none"> • Displayed but disabled • Displayed but disabled • Displayed but disabled • Not displayed or enabled 	<ul style="list-style-type: none"> • Displayed and enabled • Displayed and enabled • Displayed and enabled • Displayed but disabled • Not displayed or enabled

Configuring the Browser

This functionality is supported only if you are using the Classic Agent.

In order for Silk Test Classic to work properly, make sure that your browser is configured correctly.

If your tests use the recovery system of Silk Test Classic, that is, your tests are based on DefaultBaseState or on an application state that is ultimately based on DefaultBaseState, Silk Test Classic makes sure that your browser is configured correctly.

If your tests do not use the recovery system, you must manually configure your browser to make sure that your browser displays the following items:

- The standard toolbar buttons, for example **Home**, **Back**, and **Stop**, with the button text showing. If you customize your toolbars, then you must display at least the **Stop** button.
- The text box where you specify URLs. **Address** in Internet Explorer.
- Links as underlined text.
- The browser window's menu bar in your Web application. It is possible through some development tools to hide the browser window's menu bar in a Web application. Silk Test Classic will not work properly unless the menu bar is displayed. The recovery system cannot restore the menu bar, so you must make sure the menu bar is displayed.
- The status bar at the bottom of the window shows the full URL when your mouse pointer is over a link.

We recommend that you configure your browser to update cached pages on a frequent basis.

Internet Explorer

1. Click **Tools > Internet Options**, then click the **General** tab.
2. In the **Temporary Internet Files** area, click **Settings**.
3. On the **Settings** dialog box, select **Every visit to the page** for the **Check for newer versions of stored pages** setting.

Mozilla Firefox

1. Choose **Edit > Preferences > Advanced > Cache**.
2. Indicate when you want to compare files and update the cache. Select **Every time I view the page** at the **Compare the page in the cache to the page on the network** field.

AOL

Even though AOL's Proxy cache is updated every 24 hours, you can clear the AOL Browser Cache and force a page to reload. To do this, perform one of the following steps:

- Delete the files in the temporary internet files folder located in the Windows directory.
- Press the **CTRL** key on your keyboard and click the AOL browser reload icon (Windows PC only).

Friendly URLs

Some browsers allow you to display "friendly URLs," which are relative to the current page. To make sure you are not displaying these relative URLs, in your browser, display a page of a web site and move your mouse pointer over a link in the page.

- If the status bar displays the full URL (one that begins with the http:// protocol name and contains the site location and path), the settings are fine. For example: `http://www.mycompany.com/products.htm`
- If the status bar displays only part of the URL (for example, `products.htm`), turn off "friendly URLs." (In Internet Explorer, this setting is on the **Advanced** tab of the **Internet Options** dialog box.)

Setting Agent Options for Web Testing

This functionality is supported only if you are using the Classic Agent.

When you first install Silk Test Classic, all the options for Web testing are set appropriately. If, for some reason, for example if you were testing non-Web applications and changed them, you have problems with testing Web applications, perform the following steps:

1. Click **Options > Agent**. The **Agent Options** dialog box opens.
2. Ensure the following settings are correct.

Tab	Option	Specifies	Setting
Timing	OPT_APPREADY_TIMEOUT	The number of seconds that the agent waits for an application to become ready. Browser extensions support this option.	Site-specific; default is 180 seconds.
Timing	OPT_APPREADY_RETRY	The number of seconds that the agent waits between attempts to verify that the application is ready.	Site-specific; default is 0.1 seconds.
Other	OPT_SCROLL_INTO_VIEW	That the agent scrolls a control into view before recording events against it.	TRUE (checked); default is TRUE.
Other	OPT_SHOW_OUT_OF_VIEW	Enables Silk Test Classic to see objects not currently scrolled into view.	TRUE (checked); default is TRUE.
Verification	OPT_VERIFY_APPREADY	Whether to verify that an application is ready. Browser extensions support this option.	TRUE (checked); default is TRUE.

3. Click **OK**. The **Agent Options** dialog box closes.

Specifying a Browser for Silk Test Classic to Use in Testing a Web Application

This functionality is supported only if you are using the Classic Agent.

You can specify a browser for Silk Test Classic to use when testing a Web application at runtime or you can use the browser specified through the **Runtime Options** dialog box.

To completely automate your testing, consider specifying the browser at runtime. You can do this in one of the following ways:

- Use the `SetBrowserType` function in a script. This function takes an argument of type `BROWSERTYPE`.

- Pass an argument of type `BROWSERTYPE` to a test case as the first argument.

For an example of passing browser specifiers to a test case, see the second example in `BROWSERTYPE`. It shows you how to automate the process of running a test case against multiple browsers.

Specifying a browser through the Runtime Options dialog box

When you run a test and do not explicitly specify a browser, Silk Test Classic uses the browser specified in **Runtime Options** dialog box. To change the browser type, you can:

1. Run a series of tests with a specific browser.
2. Specify a different browser in the **Runtime Options** dialog box.
3. Run the tests again with the new browser.

Most tests will run unchanged between browsers.

Specifying your Default Browser

This functionality is supported only if you are using the Classic Agent.

Whenever you record and run test cases, you must specify the default browser that Silk Test Classic should use. If you did not choose a default browser during the installation of Silk Test Classic or if you want to change the default browser, perform the following steps:

1. Click **Options > Runtime**. The **Runtime Options** dialog box opens.
2. Select the browser that you want to use from the **Default Browser** list box.
The list box displays the browsers whose extensions you have enabled.
3. Click **OK**.

Enable Extensions Dialog Box (Classic Agent)

This functionality is supported only if you are using the Classic Agent.

Use to select the application for which you want to enable extensions. The environments listed in the dialog box are used for running the applications that you want to test. The extensions enable recognition of the non-standard controls in your environment.

You can automatically configure extensions for many development environments.

The dialog box content changes based on the agent that your project or script uses. Ensure that the agent that you want to use is selected before you open the dialog box. If necessary, close this dialog box, click the appropriate **Agent** icon in the toolbar to change the agent, and re-open the dialog box.

Click **Tools > Enable Extensions** or click **Enable Extensions** on the **Basic Workflow** bar (**Workflows > Basic**).

Silk Test Classic does not support 64-bit applications. If you are running a 64-bit application, it will display in the **Enable Extensions** dialog box, but an error occurs when you try to enable extensions. Silk Test Classic does support 32-bit applications that run in 64-bit environments and extensions will enable properly for these applications.

If you are using a project, the information you specify in the **Enable Extensions** dialog box is stored in the `partner.ini` file. If you are not using a project, the information you specify in the **Enable Extensions** dialog box is stored in the `extend.ini` file.

When you enable extensions, an include file is added based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box. For instance, if you enable extensions for Internet Explorer, Silk Test Classic adds the `explorer.inc` file to the **Runtime Options** dialog box.

Application(s) Lists all open applications that are not minimized, including any Web applications (or Java applets), which are identified by the currently loaded page's title. Click an application and then click **Select** to choose the application for which you want enable extensions. If you choose an executable name containing spaces, you must enclose the name in quotation marks.

Applets are automatically detected and can be selected from the **Application** list.

Select Selects the highlighted application. Information is gathered from the application that you select and the suggested extension settings are displayed on the **Extension Settings** dialog box.

Refresh Click to update the list of applications from which you can select.

Cancel Click to exit the dialog box without selecting an application.

Extension Information Dialog Box

Use the **Extension Information** dialog box to view information about the enabled extensions for the agent.

Click the **Classic Agent** icon in the taskbar and then click **Extensions > Details**. If the agent is not already running, click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Tools > Silk Test Classic Agent** or (in Microsoft Windows 10) **Start > Silk > Silk Test Classic Agent**.

Extension Displays the currently loaded extensions.

Status Displays the status of the currently loaded extensions. The status may be one of the following:

Loaded The extension is loaded in the application under test.

Enabled The extension is enabled, but not loaded in the application under test.

Error An error message may appear if an error has occurred.

Refresh Click to update the information that appears in the dialog box.

Extension Settings Dialog Box (.NET)

This functionality is supported only if you are using the Classic Agent.

When you select a .NET application on the **Enable Extensions** dialog box, the **Extension Settings** dialog box displays information that is specific to .NET applications.

Enable Accessibility Click to better recognize objects such as the Microsoft Office Word menu. If you are testing an application with Microsoft Accessibility objects, we recommend that you check this box. If your client/server application does not have these types of objects, you may leave the check box unchecked.

OK Click to enable the extension automatically. After the extension is enabled, Silk Test Classic can run a test to verify that the extension is working correctly.

Extension Settings Dialog Box (Web)

This functionality is supported only if you are using the Classic Agent.

When you select a Web application on the **Enable Extensions** dialog box, the **Extension Settings** dialog box displays information that is specific to Web applications. Verify the information displayed on this dialog box, and then click **OK** to enable the extension automatically. After the extension is enabled, Silk Test Classic can run a test to verify that the extension is working correctly.

Primary Extension area

Displays the extension that will be used to test your application. An extension is a file that serves to extend the capabilities of, or data available to, a more basic program. Extensions are provided for testing applications that use non-standard controls in specific development and browser environments.

DOM Enables the Document Object Model (DOM) extension which is used to query the browser directly for information about the objects on a Web page.

Secondary Extension (Select as required) area

Enable Applet Support Check if you are testing an application with applets. This check box is automatically selected when the **Enable Extensions** workflow detects an applet. You can clear the check box to prevent loading the extension. If no applet is detected, the check box is not available.

Enable Accessibility Check if you are testing an application with Microsoft Accessibility objects or other objects that may be unrecognizable. If your application does not have these types of objects, you may leave the check box unchecked.

Enable ActiveX Support Check if you are testing an application with ActiveX objects. If your application does not have these types of objects, you may leave the check box unchecked.

Extension Settings Dialog Box (Client/Server)

This functionality is supported only if you are using the Classic Agent.

When you select a client/server application on the **Enable Extensions** dialog box, the **Extension Settings** dialog box displays information that is specific to client/server applications.

Enable Accessibility Click to better recognize objects such as the Microsoft Office Word menu. If you are testing an application with Microsoft Accessibility objects, we recommend that you check this box. If your client/server application does not have these types of objects, you may leave the check box unchecked.

OK Click to enable the extension automatically. After the extension is enabled, Silk Test Classic can run a test to verify that the extension is working correctly.

Extension Settings Dialog Box (Java)

This functionality is supported only if you are using the Classic Agent.

When you select a standalone Java application on the **Enable Extensions** dialog box, the **Extension Settings** dialog box displays the path to the Java Virtual Machine (JVM), which is used by the application you have selected.

Verify that the JVM information is correct and then click **OK** to enable the extension automatically. The `SilkTest_JavaX.jar` file will be copied to the `\lib\ext` subdirectory of the JVM that the application is using. If necessary, the `accessibilities.properties` file in the `\lib` directory of this JVM will be updated or installed.

For JVM versions 1.3+, the **Copying DLLs** dialog box will display the location of the `qapjconn.dll` and `qapjarex.dll` files. During installation, these files are placed in the `Windows\System32` folder (copies

are also placed in the `SilkTest\Extend` folder). If the default directory for your library files is in a location other than `Windows\System32`, you can use the list to select the alternate location. Click **OK** to save your changes.

After the extension is enabled, a test is run to verify that the extension is working correctly.



Note:

If you defined your `CLASSPATH` in multiple set statements, it will be aggregated into a single set statement with all parameters fully expanded. Every time the `CLASSPATH` is updated, the current `autoexec.bat` is backed up and saved as `autoexec.bak`.

Understanding the Recovery System for the Open Agent

The built-in recovery system is one of the most powerful features of Silk Test Classic because it allows you to run tests unattended. When your application fails, the recovery system restores the application to a stable state, known as the BaseState, so that the rest of your tests can continue to run unattended.

The recovery system can restore your application to its BaseState at any point during test case execution:

- Before the first line of your test case begins running, the recovery system restores the application to the BaseState even if an unexpected event corrupted the application between test cases.
- During a test case, if an application error occurs, the recovery system terminates the execution of the test case, writes a message in the error log, and restores the application to the BaseState before running the next test case.
- After the test case completes, if the test case was not able to clean up after itself, for example it could not close a dialog box it opened, the recovery system restores the application to the BaseState.
- The recovery system cannot recover from an application crash that produces a modal dialog box, such as a **General Protection Fault (GPF)**.

Silk Test Classic uses the recovery system for all test cases that are based on DefaultBaseState or based on a chain of application states that ultimately are based on DefaultBaseState.

- If your test case is based on an application state of none or a chain of application states ultimately based on none, all functions within the recovery system are not called. For example, SetAppState and SetBaseState are not called, while DefaultTestCaseEnter, DefaultTestCaseExit, and error handling are called.

Such a test case will be defined in the script file as:

```
testcase Name () appstate none
```

Silk Test Classic records test cases based on DefaultBaseState as:

```
testcase Name ()
```

How the default recovery system is implemented

The default recovery system is implemented through several functions.

Function	Purpose
DefaultBaseState	Restores the default BaseState, then call the application's BaseState function, if defined.
DefaultScriptEnter	Executed when a script file is first accessed. Default action: none.
DefaultScriptExit	Executed when a script file is exited. Default action: Call the ExceptLog function if the script had errors.
DefaultTestCaseEnter	Executed when a test case is about to start. Default action: Set the application state.
DefaultTestCaseExit	Executed when a test case has ended. Default action: Call the ExceptLog function if the script had errors, then set the BaseState.

Function	Purpose
DefaultTestPlanEnter	Executed when a test plan is entered. Default action: none.
DefaultTestPlanExit	Executed when a test plan is exited. Default action: none.

You can write functions that override some of the default behavior of the recovery system.

Setting the Recovery System for the Open Agent

The recovery system ensures that each test case begins and ends with the application in its intended state. Silk Test Classic refers to this intended application state as the BaseState. The recovery system allows you to run tests unattended. When your application fails, the recovery system restores the application to the BaseState, so that the rest of your tests can continue to run unattended.

For applications that use the Open Agent and dynamic object recognition, the recovery system is configured automatically whenever the **New frame file** dialog box opens and you save a file. This dialog box opens when:

- You click **Configure Applications** on the **Basic Workflow** bar and follow the steps in the wizard.
- You click **File > New** and click **Test frame**.
- You click the **Create a new file** icon in the toolbar and then click **Test frame**.
- You click **Record > Testcase**, **Record > Application State**, or **Record > Window Locators** before you configure an application, the **New Test Frame** dialog box opens before recording starts.

If you are testing an application that uses both the Classic Agent and the Open Agent, set the Agent that will start the application as the default Agent and then set the recovery system. If you use the Classic Agent to start the application, set the recovery system for the Classic Agent.

Base State

An application's base state is the known, stable state that you expect the application to be in before each test case begins execution, and the state the application can be returned to after each test case has ended execution. This state may be the state of an application when it is first started.

Base states are important because they ensure the integrity of your tests. By guaranteeing that each test case can start from a stable base state, you can be assured that an error in one test case does not cause subsequent test cases to fail.

Silk Test Classic automatically ensures that your application is at its base state during the following stages:

- Before a test case runs.
- During the execution of a test case.
- After a test case completes successfully.

When an error occurs, Silk Test Classic does the following:

- Stops execution of the test case.
- Transfers control to the recovery system, which restores the application to its base state and logs the error in a results file.
- Resumes script execution by running the next test case after the failed test case.

The recovery system makes sure that the test case was able to "clean up" after itself, so that the next test case runs under valid conditions.

Modifying the Base State (Open Agent)

This functionality is supported only if you are using the Open Agent.

To define the state in which your application under test (AUT) should be before the execution of a test, you can change the executable location, working directory, locator, or URL of the base state. For example, if you want to launch tests from a production web site that were previously tested on a testing web site, change the base state URL and the tests are executed in the new environment.

1. Open the project for which you want to change the base state.
2. If you are testing a web application, change the browser in which the tests will be executed in the **Edit Application Configurations** dialog box.
 - a) Click **Options > Application Configurations**. The **Edit Application Configurations** dialog box appears.
 - b) Click **Edit** to the right of the application configuration that you want to change. The **Edit Browser Application Configuration** dialog box appears.
 - c) Select the browser that you want to use and click **OK**.
 - d) If you are testing a web application or a mobile native application on a remote location, for example on a mobile device that is connected to a Mac, and you want to edit the remote location, click **Change** to open the **Select Application** dialog box and then click **Edit Remote Locations**.
 - e) In the **Edit Application Configuration** dialog box, click **OK**.
3. If you are testing a web application, change the URL of the web application in the `.inc` file.
 - a) In the **File** tree, expand the **Include/Frame** node.
 - b) Double click on the `.inc` file to open the file.
 - c) Locate the URL in the file.

The URL section looks similar to the following:

```
// The start URL
const sUrl = "http://demo.borland.com/gmopost/";
```
 - d) Change the URL.
 - e) Save the `.inc` file.
4. If you are testing a desktop application, change the application configuration in the **Edit Application Configurations** dialog box.
 - a) Click **Options > Application Configurations**. The **Edit Application Configurations** dialog box appears.
 - b) Click **Edit** to the right of the application configuration that you want to change. The **Edit Windows Application Configuration** dialog box appears.
 - c) Type the executable name and file path of the desktop application that you want to test into the **Executable Pattern** text box.

For example, you might type `*\calc.exe` to specify the Calculator.
 - d) If you want to use a command line pattern in combination with the executable file, type the command line pattern into the **Command Line Pattern** text box.
5. If you are testing a mobile native application:
 - a) Click **Options > Application Configurations**. The **Edit Application Configurations** dialog box appears.
 - b) Click **Edit** to the right of the application configuration that you want to change. The **Edit Mobile Application Configuration** dialog box appears.
 - c) In the **Connection String** text box, specify the platform the location, the name of the mobile device, and the name of the mobile application that you want to test.

For example, if the device ID is 11111111, the operating system of the device is Android 5.1.1, the device is connected to the remote machine with the IP address 10.0.0.1, and you want to test the application `abc.apk`, type:

```
platformName=Android;platformVersion=5.1.1;deviceName=MotoG3;udid=11111111;host=http://10.0.0.1;app=C:\Temp\abc.apk;
```



Tip: If the application under test usually takes a long time to start, increase the application ready timeout in the replay options.

DefaultBaseState Function

Silk Test Classic provides a `DefaultBaseState` for applications, which ensures the following conditions are met before recording and executing a test case:

- The application is running.
- The application is not minimized.
- The application is the active application.
- No windows other than the application's main window are open. If the UI of the application is localized, you need to replace the strings, which are used to close a window, with the localized strings. The preferred way to replace these buttons is with the `IsCloseWindowButtons` variable in the object's declaration. You can also replace the strings in the **Close** tab of the **Agent Options** dialog box.

For Web applications that use the Open Agent, the `DefaultBaseState` also ensures the following for browsers, in addition to the general conditions listed above:

- The browser is running.
- Only one browser tab is open, if the browser supports tabs and the frame file does not specify otherwise.
- The active tab is navigated to the URL that is specified in the frame file.

For web applications that use the Classic Agent, the `DefaultBaseState` also ensures the following for browsers, in addition to the general conditions listed above:

- The browser is ready.
- Constants are set.
- The browser has toolbars, location and status bar are displayed.
- Only one tab is opened, if the browser supports tabs.

DefaultBaseState Types

Silk Test Classic includes two slightly different base state types depending on whether you use the Open Agent and dynamic object recognition or traditional hierarchical object recognition. When you use dynamic object recognition, Silk Test Classic creates a window object named `wDynamicMainWindow` in the base state. When you set the recovery system for a test that uses hierarchical object recognition, Silk Test Classic creates a window object called `wMainWindow` in the base state. Silk Test Classic uses the window object to determine which type of `DefaultBaseState` to execute.

Adding Tests that Use the Open Agent to the DefaultBaseState

If you want the recovery system to perform additional steps after it restores the default base state, record a new test case based on no application state and paste it into the declaration of the main window of your application.

1. Open your test application and the frame file of the test application.
2. Click **Record > Testcase**. Silk Test Classic displays the **Record Testcase** dialog box.
3. From the **Application state** list box, select (None).
4. Click **Start Recording**. Silk Test Classic opens the **Recording** window, which indicates that you can begin recording.
5. When you have finished recording the actions that you want to perform whenever the base state is restored, click **Stop** on the **Recording** window. Silk Test Classic displays the **Record Testcase** dialog box.
6. Click **Paste to Editor**.
7. In the **Update Files** dialog box, select **Paste testcase and update window declarations(s)**.
8. Click **OK**. Silk Test Classic creates a new script file with the new test case.
9. Add a new method named `BaseState` to the declaration of the main window in the test frame file.
10. Paste the recorded actions from the script file into the new `BaseState` method.
11. Choose **File > Save** to save the test frame file.

Examples

For example, if you want the Insurance Company web application to preselect **Auto Quote** each time the base state is restored, the new declaration for the main window in the frame file should look similar to the following:

```

window BrowserApplication WebBrowser
  locator "//BrowserApplication"

  // Go to Options -> Application Configurations... to switch
the browser
  // Alternatively set sDir and sCmdLine if you want to start a
custom browser

  // The working directory of the application when it is invoked
  // const sDir = "."

  // The command line used to invoke the application
  // const sCmdLine = ""

  // The start URL
  const sUrl = "http://demo.borland.com/InsuranceWebExtJS/
index.jsf"

  const bCloseOtherTabs = TRUE

  // The list of windows the recovery system is to leave open
  // const lwLeaveOpenWindows = {?}
  // const lsLeaveOpenLocators = {?}
  BrowserWindow BrowserWindow
    locator "//BrowserWindow"
    DomListBox QuickLinkJumpMenu
      locator "SELECT[@id='quick-link:jump-menu']"
  // ...

  // Recorded actions, which should be performed whenever the
base state of the application is restored
  Basestate()
    WebBrowser.BrowserWindow.QuickLinkJumpMenu.Select("Auto
Quote")

```

To execute the same web application on Apple Safari on an iOS device each time the base state is restored, the new declaration for the main window in the frame file should look similar to the following:

```
window BrowserApplication WebBrowser
  locator "//BrowserApplication"

  // Go to Options -> Application Configurations... to switch
  the browser
  // Alternatively set sDir and sCmdLine if you want to start a
  custom browser

  // The working directory of the application when it is invoked
  // const sDir = "."

  // The command line used to invoke the application
  // const sCmdLine = ""

  // The start URL
  const sUrl = "http://demo.borland.com/InsuranceWebExtJS/
index.jsf"

  const bCloseOtherTabs = TRUE

  // Overrides the browser type of the application configuration
  STRING sBrowserType = BROWSER_TYPE_SAFARI

  // Overrides the connection string of the application
  configuration
  STRING sConnectionString = "host=macmini1702.microfocus.com"

  // The list of windows the recovery system is to leave open
  // const lwLeaveOpenWindows = {?}
  // const lsLeaveOpenLocators = {?}
  BrowserWindow BrowserWindow
    locator "//BrowserWindow"
    DomListBox QuickLinkJumpMenu
      locator "SELECT[@id='quick-link:jump-menu']"
  // ...

  // Recorded actions, which should be performed whenever the
  base state of the application is restored
  Basestate()
  WebBrowser.BrowserWindow.QuickLinkJumpMenu.Select("Auto
Quote")
```

To execute the native mobile application InsuranceMobile.apk on an Android device each time the base state is restored, the new declaration for the main window in the frame file should look similar to the following:

```
[-] window MobileDevice Device
  [ ] locator "/MobileDevice"
  [ ]
  [ ] // Go to Options -> Application Configurations... to
  switch the mobile device
  [ ] // Alternatively set sConnectionString
  [ ]
  [ ] // Overrides the connection string of the application
  configuration
  [ ] STRING sConnectionString =
  "deviceName=MotoG3;platformName=Android;app=C:\\Temp
  \\InsuranceMobile.apk"
  [ ]
  [ ] // The list of windows the recovery system is to leave
```

```
open
  [-] MobileTextField Email
    [ ] locator "@resource-id='silktest.insurancemobile:id/
email' "
```

DefaultBaseState and the wDynamicMainWindow Object

Silk Test Classic executes the `DefaultBaseState` for dynamic object recognition when the default agent is the Open Agent and the global constant `wDynamicMainWindow` is defined. `DefaultBaseState` works with the `wDynamicMainWindow` object in the following ways:

1. If the `wDynamicMainWindow` object does not exist, invoke it, either using the `Invoke` method defined for the `MainWin` class or a user-defined `Invoke` method built into the object.
2. If the `wDynamicMainWindow` object is minimized, restore it.
3. If there are child objects of the `wDynamicMainWindow` open, close them.
4. If the `wDynamicMainWindow` object is not active, make it active.
5. If there is a `BaseState` method defined for the `wDynamicMainWindow` object, execute it.

Flow of Control

This section describes the flow of control during the execution of each of your test cases.

The Non-Web Recovery Systems Flow of Control

Before you modify the recovery system, you need to understand the flow of control during the execution of each of your test cases. The recovery system executes the `DefaultTestCaseEnter` function. This function, in turn, calls the `SetAppState` function, which does the following:

1. Executes the test case.
2. Executes the `DefaultTestCaseExit` function, which calls the `SetBaseState` function, which calls the lowest level application state, which is either the `DefaultBaseState` or any user defined application state.



Note: If the test case uses `AppState none`, the `SetBaseState` function is not called.

`DefaultTestCaseEnter()` is considered part of the test case, but `DefaultTestCaseExit()` is not. Instead, `DefaultTestCaseExit()` is considered part of the function that runs the test case, which implicitly is `main()` if the test case is run standalone. Therefore an unhandled exception that occurs during `DefaultTestCaseEnter()` will abort the current test case, but the next test case will run. However, if the exception occurs during `DefaultTestCaseExit()`, then it is occurring in the function that is calling the test case, and the function itself will abort. Since an application state may be called from both `TestCaseEnter()` and `TestCaseExit()`, an unhandled exception within the application state may cause different behavior depending on whether the exception occurs upon entering or exiting the test case.

How the Non-Web Recovery System Closes Windows

The built-in recovery system restores the base state by making sure that the non-Web application is running, is not minimized, is active, and has no open windows except for the main window. To ensure that only the main window is open, the recovery system attempts to close all other open windows, using an internal procedure that you can customize as you see fit.

To make sure that there are no application windows open except the main window, the recovery system calls the built-in `CloseWindows` method. This method starts with the currently active window and attempts to close it using the sequence of steps below, stopping when the window closes.

1. If a `Close` method is defined for the window, call it.
2. Click the **Close** menu item on the system menu, on platforms and windows that have system menus.
3. Click the window's close box, if one exists.
4. If the window is a dialog box, type each of the keys specified by the `OPT_CLOSE_DIALOG_KEYS` option and wait one second for the dialog box to close. By default, this option specifies the **Esc** key.
5. If there is a single button in the window, click that button.
6. Click each of the buttons specified by the `OPT_CLOSE_WINDOW_BUTTONS` option. By default, this option specifies the **Cancel**, **Close**, **Exit**, and **Done** keys.
7. Select each of the menu items specified by the `OPT_CLOSE_WINDOW_MENUS` option. By default, this option specifies the **File > Exit** and the **File > Quit** menu items.
8. If the closing of a window causes a confirmation dialog box to open, `CloseWindows` attempts to close the dialog box by clicking each of the buttons specified with the `OPT_CLOSE_CONFIRM_BUTTONS` option. By default, this option specifies the **No** button.

When the window, and any resulting confirmation dialog box, closes, `CloseWindows` repeats the preceding sequence of steps with the next window, until all windows are closed.

If any of the steps fails, none of the following steps is executed and the recovery system raises an exception. You may specify new window closing procedures.

In a Web application, you are usually loading new pages into the same browser, not closing a page before opening a new one.

How the Non-Web Recovery System Starts the Application

To start a non-Web application, the recovery system executes the `Invoke` method for the main window of the application. The `Invoke` method relies on the `sCmdLine` constant as recorded for the main window when you create a test frame.

For example, here is how a declaration for the `sCmdLine` constant might look for a sample Text Editor application running under Windows:

```
const sCmdLine = "c:\ProgramFiles\<<SilkTest install directory>\SilkTest\nTextEdit.exe"
```

After it starts the application, the recovery system checks whether the main window is minimized, and, if it is, uses the `Restore` method to open the icon and restore the application to its proper size.

The limit on the `sCmdLine` constant is 8191 characters.

Modifying the Default Recovery System

The default recovery system is implemented in `defaults.inc`, which is located in the directory in which you installed Silk Test Classic. If you want to modify the default recovery system, instead of overriding some of its features, as described in [Overriding the default recovery system](#), you can modify `defaults.inc`.

We cannot provide support for modifying `defaults.inc` or the results. We recommend that you do not modify `defaults.inc`. This file might change from version to version. As a result, if you manually modify `defaults.inc`, you will encounter issues when upgrading to a new version of Silk Test Classic.

If you decide to modify `defaults.inc`, be sure that you:

- Make a backup copy of the shipped `defaults.inc` file.
- Tell Technical Support when reporting problems that you have modified the default recovery system.

Overriding the Default Recovery System

The default recovery system specifies what Silk Test Classic does to restore the base state of your application. It also specifies what Silk Test Classic does whenever:

- A script file is first accessed.
- A script file is exited.
- A test case is about to begin.
- A test case is about to exit.

You can write functions that override some of the default behavior of the recovery system.

To override	Define the following
<code>DefaultScriptEnter</code>	<code>ScriptEnter</code>
<code>DefaultScriptExit</code>	<code>ScriptExit</code>
<code>DefaultTestCaseEnter</code>	<code>TestCaseEnter</code>
<code>DefaultTestCaseExit</code>	<code>TestCaseExit</code>
<code>DefaultTestPlanEnter</code>	<code>TestPlanEnter</code>
<code>DefaultTestPlanExit</code>	<code>TestPlanExit</code>

If `ScriptEnter`, `ScriptExit`, `TestcaseEnter`, `TestcaseExit`, `TestPlanEnter`, or `TestPlanExit` are defined, Silk Test Classic uses them instead of the corresponding default function. For example, you might want to specify that certain test files are copied from a server in preparation for running a script. You might specify such processing in a function called `ScriptEnter` in your test frame.

If you want to modify the default recovery system, instead of overriding some of its features, you can modify `defaults.inc`. We do not recommend modifying `defaults.inc` and cannot provide support for modifying `defaults.inc` or the results.

Example

If you are planning on overriding the recovery system, you need to write your own `TestCaseExit(Boolean bException)`. In the following example, `DefaultTestcaseExit()` is called inside `TestCaseExit()` to perform standard recovery systems steps and the `bException` argument is passed into `DefaultTestCaseExit()`.

```
if (bException)
    DefaultTestcaseExit(bException)
```

If you are not planning to call `DefaultTestcaseExit()` and plan to handle the error logging in your own way, then you can use the `TestcaseExit()` signature without any arguments.

Use the following function signature if you plan on calling `DefaultTestcaseExit(Boolean bException)` or if your logic depends on whether an exception occurred. Otherwise, you can simply use the function signature of `TestcaseExit()` without any arguments. For example, the following is from the description of the `ExceptLog()` function.

```
TestCaseExit (BOOLEAN bException)
if (bException)
    ExceptLog()
```

Here, `DefaultTestcaseExit()` is not called, but the value of `bException` is used to determine if an error occurred during the test case execution.

Handling Login Windows

Silk Test Classic handles login windows differently, depending on whether you are testing Web or client/server applications. These topics provide information on how to handle login windows in your application under test.

Handling Login Windows in Non-Web Applications that Use the Open Agent

Although a non-Web application's main window is usually displayed first, it is also common for a login or security window to be displayed before the main window.

Use the `wStartup` constant and the `Invoke` method

To handle login windows, record a declaration for the login window, set the value of the `wStartup` constant, and write a new `Invoke` method for the main window that enters the appropriate information into the login window and dismisses it. This enables the `DefaultBaseState` routine to perform the actions necessary to get past the login window.

You do not need to use this procedure for splash screens, which disappear on their own.

1. Open the login window that precedes the application's main window.
2. Open the test frame.
3. Click **Record > Window Locators** to record a locator for the window.
4. Point to the title bar of the window and then press **Ctrl+Alt**. The locator is captured in the **Record Window Locators** dialog box.
5. Click **Paste to Editor** to paste the locator into the test frame.
6. In the **Record Window Locators** dialog box, click **Close**.
7. Close your application.
8. In your test frame file, find the stub of the declaration for the `wStartup` constant, located at the top of the declaration for the main window:

```
// First window to appear when application is invoked
// const wStartup = ?
```

9. Complete the declaration for the `wStartup` constant by:
 - Removing the comment characters, the two forward slash characters, at the beginning of the declaration.
 - Replacing the question mark with the identifier of the login window, as recorded in the window declaration for the login window.
10. Define an `Invoke` method in the main window declaration that calls the built-in `Invoke` method and additionally performs any actions required by the login window, such as entering a name and password.

After following this procedure, your test frame might look like this:

```
window MainWin MyApp
  locator "/MainWin[@caption='MyApp']"
  const wStartup = Login

  // the declarations for the MainWin should go here
  Invoke ()
    derived::Invoke ()
    Login.Name.SetText ("Your name")
    Login.Password.SetText ("password")
    Login.OK.Click ()

window DialogBox Login
  locator "/DialogBox[@caption='Login']"
```

```
// the declarations for the Login window go here
PushButton OK
locator "OK"
```



Note: Regarding the derived keyword and scope resolution operator. The statement `derived::Invoke ()` uses the derived keyword followed by the scope resolution operator `(::)` to call the built-in `Invoke` method, before performing the operations needed to fill in and dismiss the login window.

Specifying Windows to be Left Open for Tests that Use the Open Agent

By default, the non-Web recovery system closes all windows in your test application except the main window. To specify which windows, if any, need to be left open — such as a child window that is always open — use the `lwLeaveOpenWindows` or `lsLeaveOpenLocators` constant.

`lwLeaveOpenWindows` and `lsLeaveOpenLocators` constants

When you record and paste the declarations for your application's main window, the stub of a declaration for the `lwLeaveOpenWindows` constant is automatically included. Additionally, it is possible to specify windows to leave open by using XPath locator strings. These can be specified with the variable `lsLeaveOpenLocators`, which must be a list of strings. The following example shows the `lwLeaveOpenWindows` and `lsLeaveOpenLocators` constants before they have been edited:

```
// The list of windows the recovery system is to leave open
// const lwLeaveOpenWindows = {?}
// const lsLeaveOpenLocators = {?}

```

To complete the declaration for these constants:

1. For `lwLeaveOpenWindows`, replace the question mark in the comment with the 4Test identifiers of the windows you want to be left open. Separate each identifier with a comma.
2. For `lsLeaveOpenLocators`, click **Record > Window Locators** and record the locators that you want to include.
3. Replace the question mark in the comment with the locator strings for the windows that you want to be left open. Separate each identifier with a comma.
4. Remove the comment characters (the two forward slash characters) at the beginning of the `lwLeaveOpenWindows` declaration.

For example, the following code shows how to set the `lwLeaveOpenWindows` constant so that the recovery system leaves open the window with the identifier `DocumentWindow` when it restores the `BaseState`.

```
const lwLeaveOpenWindows = {DocumentWindow}
```

5. Remove the comment characters (the two forward slash characters) at the beginning of the `lsLeaveOpenLocators` declaration.

For example, the following code shows how to set the `lsLeaveOpenLocators` constant so that the recovery system leaves open the **About** dialog box when it restores the `BaseState`.

```
lsLeaveOpenLocators = {"/MainWin[@caption='*Information*']", "//
Dialog[@caption='About']"}
```

Specifying New Window Closing Procedures

When the recovery system cannot close a window using its normal procedure, you can reconfigure it in one of two ways:

- If the window can be closed by a button press, key press, or menu selection, specify the appropriate option either statically in the **Close** tab of the **Agent Options** dialog box or dynamically at runtime.
- Otherwise, record a `Close` method for the window.

This is only for classes derived from the `MoveableWin` class: `DialogBox`, `ChildWin`, and `MessageBox`. Specifying window closing procedures is not necessary for web pages, so this does not apply to `BrowserChild` objects/classes.

Specifying Buttons, Keys, and Menus that Close Windows

Specify statically

To specify statically the keys, menu items, and buttons that the non-Web recovery system should use to close all windows, choose **Options > Agent** and then click the **Close** tab.

The **Close** tab of the **Agent Options** dialog box contains a number of options, each of which takes a comma-delimited list of character string values.

Specify dynamically

As you set close options in the **Agent Options** dialog box, the informational text at the bottom of the dialog box shows the 4Test command you can use to specify the same option from within a script; add this 4Test command to a script if you need to change the option dynamically as a script is running.

Specify for individual objects

If you want to specify the keys, menu items, and buttons that the non-web recovery system should use to close an individual dialog box, define the appropriate variable in the window declaration for the dialog box:

- `lsCloseWindowButtons`
- `lsCloseConfirmButtons`
- `lsCloseDialogKeys`
- `lsCloseWindowMenus`

This is only for classes derived from the `MoveableWin` class: `DialogBox`, `ChildWin`, and `MessageBox`. Specifying window closing procedures is not necessary for web pages, so this does not apply to `BrowserChild` objects/classes.

Recording a Close Method for Tests that Use the Open Agent

To specify the keys, menu items, and buttons that the non-web recovery system uses to close an individual dialog box, record a `Close` method to define the appropriate variable in the window declaration for the dialog box.

1. Open your application.
2. Open the application's test frame file.
3. Choose **Record > Testcase**. Silk Test Classic displays the **Record Testcase** dialog box.
4. From the **Application state** list box, click `(None)`.
5. Click **Start Recording**. Silk Test Classic opens the **Recording** window, which indicates that you can begin recording the `Close` method.
6. When you have finished recording the `Close` method, click **Stop** on the **Recording** window. Silk Test Classic redisplay the **Record Testcase** dialog box.

7. Click **Paste to Editor** and then copy and paste the script in the declaration for the dialog box in the test frame file.
8. Choose **File > Save** to save the test frame file.

You can also specify buttons, keys, and menus that close windows. This is only for classes derived from the `MoveableWin` class: `DialogBox`, `ChildWin`, and `MessageBox`. Specifying window closing procedures is not necessary for Web pages, so this does not apply to `BrowserChild` objects/classes.

Set Recovery System Dialog Box

Use the **Set Recovery System** dialog box to identify the starting point of the application you are testing, the `BaseState`. The recovery system will return your application to this `BaseState`:

- Before running a test case.
- During a test case.
- If an error occurs.
- After a test case completes.

If you are using the **Basic workflow** bar, click **Set Recovery System**.

If you are recording a test case, click **Set Recovery System** on the **Record Application State** dialog box or the **Record Testcase** dialog box.

Frame filename Displays the default name and path of the frame file you are creating. This field appears only if you access this dialog box from the **Basic workflow** bar. The default is `frame.inc`. If `frame.inc` already exists, Silk Test appends the next logical number to the new frame file name. For example, `frame1.inc`.

Modify the frame file name and click **Browse** to specify the location in which you want to save this file. Frame files must have a `.inc` extension.

Application Lists all open applications that are not minimized, including any Web applications, which are identified by the title of the currently loaded page. Click to select an application. This list is dynamic and will update if you open a new application.

If you are connected to the Open Agent, only those applications that have extensions enabled display in the list.

Command line Displays the path to the executable (`.exe`) for the application that you selected. This field does not display if you have selected a Web application.

Start testing on this page Displays the URL for the application you selected. This field displays only if you have selected a Web application. If an application displays in the list, but the URL does not display in this field, your extensions may not be enabled correctly. Click the **Enable Extensions** button in the **Basic workflow** bar to automatically enable and test extension settings.

Working directory Displays the path of the application you selected. This field does not display if you have selected a Web application.

Window name Displays the window name, a suggested identifier that you can use in your test cases to identify your application. You can change the window name. We recommend using a short name to identify your application.

Test Plans

A test plan usually is a hierarchically-structured document that describes the test requirements and contains the statements, 4Test scripts, and test cases that implement the test requirements. A test plan is displayed in an easy-to-read outline format, which lists the test requirements in high-level prose descriptions. The structure can be flat or many levels deep.

Indentation and color indicate the level of detail and various test plan elements. Large test plans can be divided into a master plan and one or more sub-plans. A test plan file has a .pln extension, such as `find.pln`.

Structuring your test plan as an hierarchical outline provides the following advantages:

- Assists the test plan author in developing thoughts about the test problem by promoting and supporting a top-down approach to test planning.
- Yields a comprehensive inventory of test requirements, from the most general, through finer and finer levels of detail, to the most specific.
- Allows the statements that actually implement the tests to be shared by group descriptions or used by just a single test description.
- Provides reviewers with a framework for evaluating the thoroughness of the plan and for following the logic of the test plan author.
- If you are using the test plan editor, the first step in creating automated tests is to create a test plan. If you are not using the test plan editor, the first step is creating a test frame.

Structure of a Test Plan

A test plan is made up of the following elements, each of which is identified by color and indentation on the test plan.

Element	Description	Color
Comment	Provide documentation throughout the test plan; preceded by <code>//</code> .	Green
Group Description	High level line in the test requirements outline that describes a group of tests.	Black
Test Description	Lowest level line describing a single test case; is a statement of the functionality to be tested by the associated test case.	Blue
Test Plan Statement	Used to provide script name, test case name, test data, or <code>include</code> statement.	Red when a sub plan is not expanded. Magenta statement when sub-plan is expanded

A statement placed at the group description level applies to all the test descriptions contained by the group. Conversely, a statement placed at the test description level applies only to that test description. Levels in the test plan are represented by indentation.

Because there are many ways to organize information, you can structure a test plan using as few or as many levels of detail as you feel are necessary. For example, you can use a list structure, which is a list of test descriptions with no group description, or a hierarchical structure, which is a group description and test description. The goal when writing test plans is to create a top-down outline that describes all of the test requirements, from the most general to the most specific.

Overview of Test Plan Templates

Because a test plan is initially empty, you may want to insert a template, which is a hierarchical outline you can use as a guide when you create a new test plan.

The template contains placeholders for each GUI object in your application. Although you may not want to structure the test plan in a way which mirrors the hierarchy of your application's GUI, this can be a good starting point if you are new to creating test plans.

In order to be able to insert a template, you must first record a test frame, which contains declarations for each of the GUI objects in your application.

Example Outline for Word Search Feature

Because a test plan is made up of a large amount of information, a structured, hierarchical outline provides an ideal model for organizing and developing the details of the plan. You can structure an outline using as few or as many levels of detail as you feel necessary.

The following is a series of sample outlines, ranging from a simple *list structure* to a more specific *hierarchical structure*. For completeness, each of the plans also shows the script and test case statements that link the descriptions to the 4Test scripts and test cases that implement the test requirements.

For example, consider the **Find** dialog box from the Text Editor application, which allows a user to search in a document. A user enters the characters to search for in the **Find What** text box, checks the **Case sensitive** check box to consider case, and clicks either the **Up** or **Down** radio button to indicate the direction of the search.

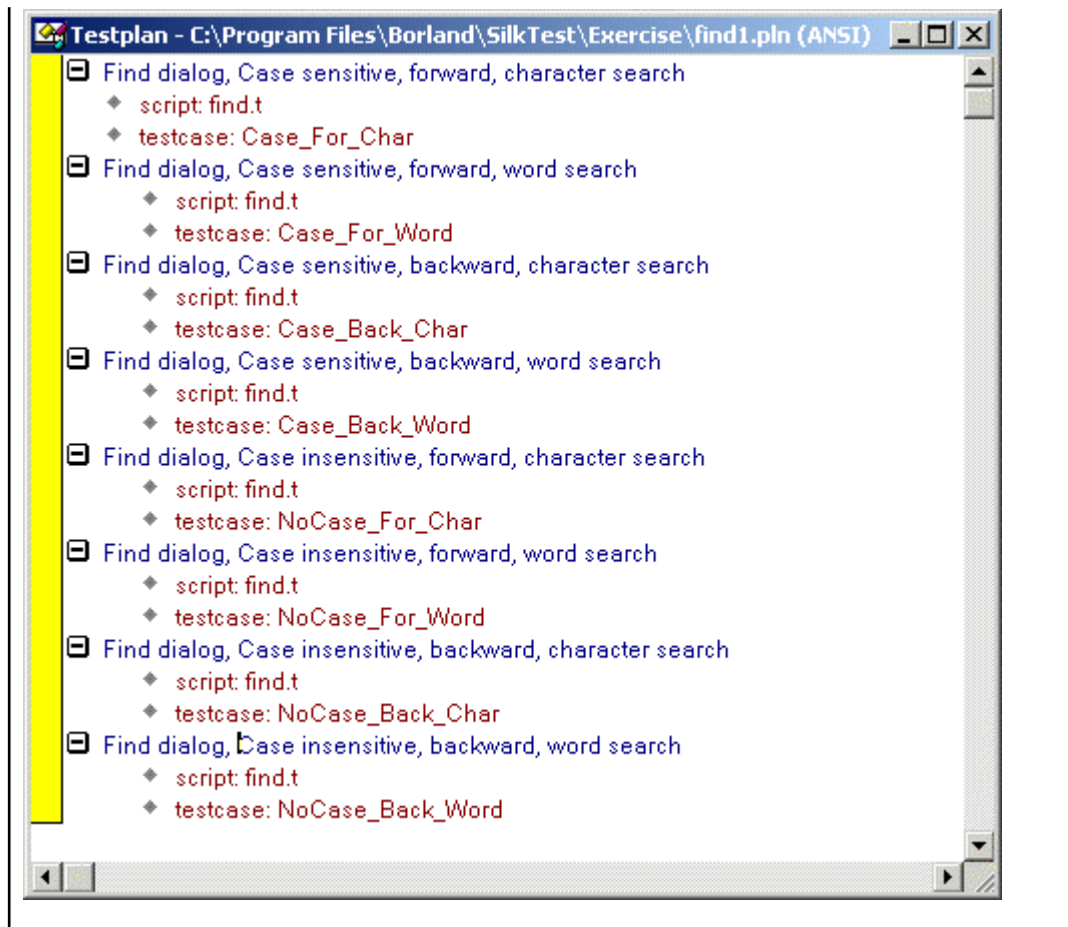
List Structure

At its simplest, an outline is a hierarchy with just a single level of detail. In other words, it is a list of test descriptions, with no group descriptions.

Using the list structure, each test is fully described by a single line, which is followed by the script and test case that implement the test. You may find this style of plan useful in the beginning stages of test plan design, when you are brainstorming the list of test requirements, without regard for the way in which the test requirements are related. It is also useful if you are creating an ad hoc test plan that runs a set of unrelated 4Test scripts and test cases.

Example for List Structure

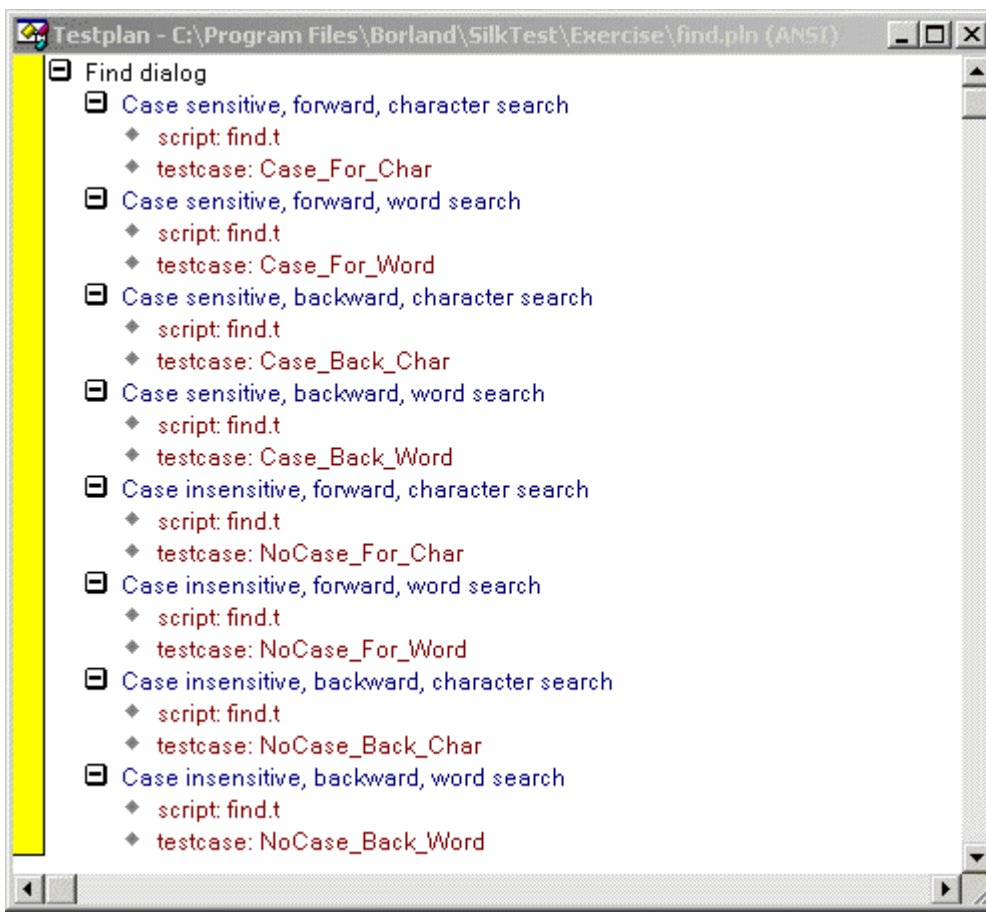
For example:



Hierarchical Structure

The following test plan has a single level of group description, preceding the level that contains each of the test descriptions. The group description indicates that all the tests are for the **Find** dialog box.

As the figure shows, the test plan editor indicates levels in the outline with indentation. Each successive level is indented one level to the right. The minus icons indicate that each of the levels is fully expanded. By clicking on the minus icon at any level, you collapse the branch below that level. When working with large test plans, collapsing and expanding test plan detail makes it easy to see as much or as little of the test plan as you need. You could continue this test plan by adding a second level of group description, indicating whether or not the tests in the group are case sensitive, and even more detail by adding a third level of group descriptions which indicate whether the tests in the group search in the forward or backward direction.



Converting a Results File to a Test Plan

You can use the **Convert Results to Plan** dialog box to transform a results file to a test plan.

1. Open a results file that was generated by running a script file.
2. Click **Results > Convert to Plan**. The **Convert Results to Plan** dialog box appears.
3. Select the results file that you want to convert.
4. Click **OK**.

When creating a test plan from a results file generated for a script, the test plan editor uses the # symbol so that when this test plan is run, the `testdata` statement doubles as description. Since the results file was for a script, not a test plan, it does not contain any group or test case descriptions. The # symbol can be used with any test plan editor statement so that the statement will double as description.

Working with Test Plans

This section describes how you can work with test plans.

Creating a New Test Plan

1. Click **File > New**.
2. Click **Test plan** and click **OK**. An empty test plan window opens.

3. Create your test plan and then click **File > Save**.
4. Specify the name and location in which to save the file, and then click **OK**.
5. If you are working within a project, Silk Test Classic prompts you to add the file to the project. Click **Yes** if you want to add the file to the open project, or **No** if you do not want to add this file to the project.

Before you can begin testing, you must enable extensions for applications you plan to test on both the target machine and the host machine.

Indent and Change Levels in an Outline

You can use menu, keyboard, or toolbar commands to enter or change group and test descriptions as you are typing them. The following table summarizes the commands:

Action	Menu Item	Key
Indent one level	Outline/Move Right	<u>ALT + forward arrow</u>
Outdent one level	Outline/Move Left	<u>ALT + back arrow</u>
Swap with line above	Outline/Transpose Up	<u>ALT + up arrow</u>
Swap with line below	Outline/Transpose Down	<u>ALT + down arrow</u>

Each command acts on the current line or currently selected lines.

Silk Test Classic ignores comments when compiling, with the exception of functions and test cases. Comments within functions and test cases must be within the scope of the function/test case. If a comment is outdented beyond the scope of the function/test case, the compiler assumes that the function/test case has ended. As long as comments do not violate the function/test case scope, they can be placed anywhere on a line.



Note: Comments beyond the scope can also impact expand/collapse functionality and may prevent a function/test case from being fully expanded/collapsed. We recommend that you keep comments within scope.

Adding Comments to Test Plan Results

You can add comments to your test plans which will display in the results when you run your tests. You can annotate your tests with such comments to ease the interpretation of the test results.

To add a comment to a test plan, include the following statement in the test plan:

```
comment: Your comment text
```

For example, running the following piece of a test plan:

```
Find dialog
  Get the default button
    comment: This test should return Find.FindNext
    script: find.t
    testcase: GetButton
```

produces the following in the results file:

```
Find dialog
  Get the default button
    Find.FindNext
    comment: This test should return Find.FindNext
```



Note: You can also preface lines in all 4Test files with // to indicate a single-line comment. Such comments do not display in test plan results.

Documenting Manual Tests in the Test Plan

Your QA department might do some of its testing manually. You can document the manual testing in the test plan. In this way, the planning, organization, and reporting of all your testing can be centralized in one place. You can describe the state of each of your manual tests. This information is used in reports.

To indicate that a test description in the test plan is implemented with a manual test, use the value `manual` in the `testcase` statement, as in:

```
testcase: manual
```

By default, whenever you generate a report, it includes information on the tests run for that results file, plus the current results of any manual tests specified in the test plan. If the manual test results are subsequently updated, the next time you generate the report, it incorporates the latest manual results. However, this might not be what you want. If you want the report to use a snapshot of manual results, not the most recent manual results, merge the results of manual tests into the results file.

Describing the State of a Manual Test

1. Open a test plan containing manual tests.
2. Click **Testplan > Run Manual Tests**.
3. Select a manual test from the **Update Manual Tests** dialog box and document it. The **Update Manual Tests** dialog box lists all manual tests in the current test plan.

Mark the test complete

Click the **Complete** option button.

`Complete` means that a test has been defined. A manual test marked here as `Complete` will be tabulated as `complete` in `Completion` reports.

Indicate whether the test passed or failed

1. Click the **Has been run** option button.
2. Select **Passed** or **Failed**.
3. Specify when the test was run and optionally, specify the machine.

To specify when the test was run, use the following syntax:

```
YYYY-MM-DD HH:MM:SS
```

Hours, minutes, and seconds are optional. For example, enter `2006-01-10` to indicate that the test was run Jan 10, 2006.

Manual tests marked as `Passed` or `Failed` will be tabulated as such in `Pass/Fail` reports, as long as you have also specified the time at which they were run.

A test marked `Has been run` is also considered `complete` in `Completion` reports.

Add any comments you want about the test

Fill in the **Comments** text box.

Inserting a Template

1. Click **Testplan > Insert Template**. The **Insert Testplan Template** dialog box, which lists all the GUI objects declared in your test frame, opens.
2. Select each of the GUI objects that are related to the application features you want to test. Because this is a multi-select list box, the objects do not have to be contiguous. For each selected object, Silk Test Classic inserts two lines of descriptive text into the test plan.

For example, the test plan editor would create the following template for the **Find** dialog box of the Text Editor application:

```

Tests for DialogBox Find
Tests for StaticText FindWhatText
(Insert tests here)
Tests for TextField FindWhat
(Insert tests here)
Tests for CheckBox CaseSensitive
(Insert tests here)
Tests for StaticText DirectionText
(Insert tests here)
Tests for PushButton FindNext
(Insert tests here)
Tests for PushButton Cancel
(Insert tests here)
Tests for RadioList Direction
(Insert tests here)

```

Changing Colors in a Test Plan

You can customize your test plan so that different test plan components display in unique colors.

To change the default colors:

1. Click **Options > Editor Colors**.
2. On the **Editor Colors** dialog box, select the outline editor item you want to change in the **Editor Item** list box at the left of the dialog box.
3. Apply a color to the item by selecting a pushbutton from the list of predefined colors or create a new color to apply by selecting the red, green, and blue values that compose the color.

Default color	Component	Description
Blue	Test description	Lowest level of the hierarchical test plan outline that describes a single test case.
Red	Test plan statement	Link scripts, test cases, test data, closed sub-plans, or an include file, such as a test frame, to the test plan.
Magenta	Include statement when sub-plan is open	Sub-plans to be included in a master plan.
Green	Comment	Additional user information that is incidental to the outline; preceded by double slashes (//); provides documentation throughout the test plan.
Black	Other line (group description)	Higher level lines of the hierarchical test plan outline that describe a group of tests; may be several levels in depth.

Linking the Test Plan to Scripts and Test Cases

After you create your test plan, you can associate the appropriate 4Test scripts and test cases that implement your test plan. You create this association by inserting `script` and `testcase` statements in the appropriate locations in the test plan.

There are three ways to link a script or test case to a test plan:

- Linking a description to a script or test case using the **Testplan Detail** dialog box if you want to automate the process of linking scripts and test cases to the test plan.
- Linking to a test plan manually.
- Linking scripts and test cases to a test plan: the test plan editor automatically inserts the `script` and `testcase` statements into the plan once the recording is finished, linking the plan to the 4Test code.

You can insert a `script` and `testcase` statement for each test description, although placing a statement at the group level when possible eliminates redundancy in the test plan. For example, since it is usually

good practice to place all the test cases for a given application feature into a single script file, you can reduce the redundancy in the test plan by specifying the `script` statement at the group level that describes that feature.

You can also insert a `testcase` statement at the group level, although doing so is only appropriate when the test case is data driven, meaning that it receives test data from the plan. Otherwise the same test case would be called several times with no difference in outcome.

Insert Testplan Template Dialog Box

Use the **Insert Testplan Template** dialog box to insert a hierarchical outline (template) of objects into your testplan. This dialog is only available when a testplan is open and the Testplan dialog is active. In order to be able to insert a template, you must first record a test frame, which contains declarations for each of the objects in your application.

Click **Testplan > Insert Template**.

Windows to create a template for Displays a list of all the objects declared in your test frame. Select objects related to the application features you want to test in your testplan. (Press **Ctrl-Click** to select multiple objects.) For each selected object, two lines of descriptive text are inserted into the testplan.

Working with Large Test Plans

For large or complicated applications, the test plan can become quite large. This raises the following issues:

Issue	Solution
How to keep track of where you are in the test plan and what is in scope at that level.	Use the Testplan Detail dialog box.
How to determine which portions of the test plan have been implemented.	Produce a Completion report.
How to allow several staff members to work on the test plan at the same time.	Structure your test plan as a master plan with one or more sub-plans.

This section describes how you can divide your test plan into a master plan with one or more sub-plans to allow several staff members to work on the test plan at the same time.

Determining Where Values are Defined in a Large Test Plan

1. Place the insertion point at the relevant point in the test plan and click **Testplan > Detail**. The **Testplan Detail** dialog box opens.
2. Click the level in the list box at the top of the **Testplan Detail** dialog box, to see just the set of symbols, attributes, and statements that are defined on a particular level.
3. Once you find the level at which a symbol, attribute, or statement was defined, you can change the value at that level, causing the inherited value at the lower levels to change also.

Dividing a Test Plan into a Master Plan and Sub-Plans

If several engineers in your QA department will be working on a test plan, it makes sense to break up the plan into a master plan and sub-plans. This approach allows multi-user access, while at the same time maintaining a single point of control for the entire project.

The master plan contains only the top few levels of group descriptions, and the sub-plans contain the remaining levels of group descriptions and test descriptions. Statements, attributes, symbols, and test data defined in the master plan are accessible within each of the sub-plans.

Sub-plans are specified with an `include` statement. To expand the sub-plan files so that they are visible within the master plan, double-click in the left margin next to the `include` statement. Once a sub-plan is expanded inline, the sub-plan statement changes from red (the default color for statements) to magenta, indicating that the line is now read-only and that the sub-plan is expanded inline. At the end of the expanded sub-plan is the `<eof>` marker, which indicates the end of the sub-plan file.

Creating a Sub-Plan

You create a sub-plan in the same way you create any test plan: by opening a new test plan file and entering the group descriptions, test descriptions, and the test plan editor statements that comprise the sub-plan, either manually or using the **Testplan Detail** dialog.

Copying a Sub-Plan

When you copy and paste the include statement and the contents of an open include file, note that only the include statement will be pasted.

To view the contents of the sub-plan, open the pasted include file by clicking **Include > Open** or double-click the margin to the left of the include statement.

Opening a Sub-Plan

Open the sub-plan from within the master plan. To do this, you can either:

- double-click the margin to the left of the include statement or
- highlight the include statement and choose **Include > Open**. (Compiling a script also automatically opens all sub-plans.)

If a sub-plan does not inherit anything (that is, statements, attributes, symbols, or data) from the master plan, you can open the sub-plan directly from the **File > Open** dialog box.

Connecting a Sub-Plan with a Master Plan

To connect the master plan to a sub-plan file, you enter an `include` statement in the master plan at the point where the sub-plan logically fits. The `include` statement cannot be entered through the **Testplan Detail** dialog box; you must enter it manually.

The `include` statement uses this syntax:

```
include: myinclude.pln
```

where `myinclude` is the name of the test plan file that contains the sub-plan.

If you enter the `include` statement correctly, it displays in red, the default color used for the test plan editor statements. Otherwise, the statement displays in blue or black, indicating a syntax error (the compiler is interpreting the line as a description, not a statement).

Refreshing a Local Copy of a Sub-Plan

When another user modifies a sub-plan, those changes are not automatically reflected in your read-only copy of the sub-plan. Once the other user has released the lock on the sub-plan, there are two ways to refresh your copy:

1. Close and then reopen the sub-plan.

2. Acquire a lock for the sub-plan.

Sharing a Test Plan Initialization File

All QA engineers working on a test plan that is broken up into a master plan and sub-plans must use the same test plan initialization file.

To share a test plan initialization file:

1. Click **Options > General**.
2. On the **General Options** dialog box, specify the same file name in the **Data File for Attributes and Queries** text box.

Saving Changes

When you finish editing, choose **Include > Save** to save the changes to the sub-plan.

Include > Save saves changes to the current sub-plan while **File > Save** saves all open master plans and sub-plans.

Overview of Locks

When first opened, a master plan and its related sub-plans are read-only. This allows many users to open, read, run, and generate reports on the plan. When you need to edit the master plan or a sub-plan, you must first acquire a lock, which prevents others from making changes that conflict with your changes.

Acquiring and Releasing a Lock

Acquire a lock Place the cursor in or highlight one or more sub-plans and then choose **Include > Acquire Lock**.

The bar in the left margin of the test plan changes from gray to yellow.

Release a lock Select **Include > Release Lock**.

The margin bar changes from yellow to gray.

Generating a Test Plan Completion Report

To measure your QA department's progress in implementing a large test plan, you can generate a completion report. The completion report considers a test complete if the test description is linked to a test case with two exceptions:

- If the test case statement invokes a data-driven test case and a symbol being passed to the data-driven test case is assigned the value ? (undefined), the test is considered incomplete.
- If the test case is manual and marked as Incomplete in the **Update Manual Tests** dialog box, the test is considered incomplete. A manual test case is indicated with the `testcase:manual` syntax.

To generate a test plan completion report:

1. Open the test plan on which you want to report.
2. Click **Testplan > Completion Report** to display the **Testplan Completion Report** dialog box.
3. In the **Report Scope** group box, indicate whether the report is for the entire plan or only for those tests that are marked.
4. To subtotal the report by a given attribute, select an attribute from the **Subtotal by Attribute** text box.
5. Click **Generate**.

The test plan editor generates the report and displays it in the lower half of the dialog box. If the test plan is structured as a master plan with associated sub-plans, the test plan editor opens any closed sub-plans before generating the report.

You can:

- Print the report.
- Export the report to a comma-delimited ASCII file. You can then bring the report into a spreadsheet application that accepts comma-delimited data.
- Chart (graph) the report, just as you can chart a Pass/Fail report.

Testplan Completion Report Dialog Box

Use the **Testplan Completion Report** dialog box to generate a report on the number of completed tests. This dialog is only available when a test plan is open and the **Testplan** dialog box is active.

To open the **Testplan Completion Report** dialog box, click **Testplan > Completion Report**.

The **Testplan Completion Report** considers a test complete if the test description is linked to a test case, with two exceptions:

- If the testcase statement invokes a data-driven test case and a symbol being passed to the data-driven test case has the value ? (undefined) assigned, the test is considered incomplete.
- If the test case is manual and has not been marked as complete with the **Finish Test Run** button on the **Execute Manual Test** dialog box, the test is considered incomplete.

Test plan items that are marked with attributes of type `set` are not categorized in a **Testplan Completion Report**.

Marked tests	Select to generate a report only for those tests that are marked within the test plan.
All tests	Select to generate a report for all tests within the test plan.
Subtotal by attribute	If you want to subtotal the report by a specific attribute, select an attribute from the list. Default attributes are <i>Category</i> , <i>Component</i> , and <i>Developer</i> .
Display Percentages	Check this check box to display completion percentages for the selected test plan. Click the Chart tab to display this check box.
Report tab	Displays the completion report for the selected test plan.
Chart tab	Displays the completion chart for the selected test plan.
Print	Click to print the selected test plan completion report or chart. Click the appropriate tab, report or chart, and then click Print .
Export	Click to export the report as an ANSI, Unicode, or UTF-8 file.
Generate	Click to generate a Testplan Completion Report and Chart , based on the options you specified, for the selected test plan.
Setup	Click to display the Chart Settings dialog box where you can modify the format in which test plan completion charts display.

Adding Data to a Test Plan

This section describes how you can add data to a test plan.

Specifying Unique and Shared Data

If a data value is unique to a single test description

You should place it in the plan at the same level as the test description, using the `testdata` statement. You can add the `testdata` statement using the **Testplan Detail** dialog box or type the `testdata` statement directly into the test plan.

If data is common to several tests

You can factor out the data that is common to a group of tests and define it at a level in the test plan where it can be shared by the group. To do this, you define symbols and assign them values. Using symbols results in less redundant data, and therefore, less maintenance.

Adding Comments in the Test Plan Editor

Use two forward slash characters to indicate that a line in a test plan is a comment. For example:

```
// This is a comment
```

Comments preceded by `//` do not display in the results file. You can also specify comments using the comment statement; these comments will display in the results files.

Testplan Editor Statements

You use the test plan editor keywords to construct statements, using this syntax:

```
keyword : value
```

`keyword`: One of the test plan editor keywords.

`value`: A comment, script, test case, include file, attribute name, or data value.

For example, this statement associates the script `myscript.t` with the plan:

```
script : myscript.t
```

Spaces before and after the colon are optional.

The # Operator in the Testplan Editor

When a `#` character precedes a statement, the statement will double as a test description in the test plan. This helps eliminate possible redundancies in the test plan. For example, the following test description and script statement:

```
Script is test.t
    script:test.t
```

can be reduced to one line in the test plan:

```
#script: test.t
```

The test plan editor considers this line an executable statement as well as a description. Any statements that follow this "description" in the test plan and that trigger test execution must be indented.

Using the Testplan Detail Dialog Box to Enter the testdata Statement

1. Place the insertion point at the end of the test description. If a `testdata` statement is not associated with a test description, the compiler generates an error.
2. Click **Testplan > Detail**. To provide context, the multi-line list box at the top of the **Testplan Detail** dialog box displays the line in the test plan that the cursor was on when the dialog box was invoked,

indicated by the black arrow icon. If the test case and script associated with the current test description are inherited from a higher level in the test plan, they are shown in blue; otherwise, they are shown in black.

3. Enter the data in the **Test Data** text box, separating each data element with a comma.

Remember, if the test case expects a record, you need to enclose the list of data with the list constructor operator (the curly braces); otherwise, Silk Test Classic interprets the data as individual variables, not a record, and will generate a data type mismatch compiler error.

4. Click **OK**. Silk Test Classic closes the **Testplan Detail** dialog box and enters the `testdata` statement and data values in the plan.

Entering the `testdata` Statement Manually

1. Open up a new line after the test description and indent the line one level.

2. Enter the `testdata` statement as follows.

- If the test case expects one or more variables, use this syntax: `testdata: data [,data]`, where `data` is any valid 4Test expression.
- A record, use the same syntax as above, but open and close the list of record fields with curly braces: `testdata: {data [,data]}`, where `data` is any valid 4Test expression.

Be sure to follow the `testdata` keyword with a colon. If you enter the keyword correctly, the statement displays in dark red, the default color. Otherwise, the statement displays in either blue or black, indicating the compiler is interpreting the line as a description.

Linking Test Plans

This section describes how Silk Test Classic handles linking from a test plan to a script or test case.

Linking a Description to a Script or Test Case using the Testplan Detail Dialog Box

1. Place the insertion cursor on either a test description or a group description.

2. Click **Testplan > Detail**. The test plan editor invokes the **Testplan Detail** dialog box, with the **Test Execution** tab showing. The multi-line list box at the top of the dialog box displays the line in the test plan that the cursor was on when the dialog box was invoked, as well as its ancestor lines. The black arrow icon indicates the current line. The current line appears in black and white, and the preceding lines display in blue.

3. If you:

- know the names of the script and test case, enter them in the **Script** and **Testcase** fields, respectively.
- are unsure of the script name, click the **Scripts** button to the right of the **Script** field to browse for the script file.

4. On the **Testplan Detail - Script** dialog box, navigate to the appropriate directory and select a script name by double-clicking or by selecting and then clicking **OK**. Silk Test Classic closes the **Testplan Detail - Script** dialog box and enters the script name in the **Script** field.

5. Click the **Testcases** button to the right of the **Testcase** field, to browse for the test case name.

The **Testplan Detail – Testcase** dialog box shows the names of the test cases that are contained in the selected script. Test cases are listed alphabetically, not in the order in which they occur in the script.

6. Select a test case from the list and click **OK**.

7. Click **OK**. The script and test case statements are entered in the plan.

If you feel comfortable with the syntax of the test plan editor statements and know the locations of the appropriate script and test case, you can enter the script and test case statements manually.

Linking a Test Plan to a Data-Driven Test Case

To link a group of test descriptions in the plan with a data-driven test case, add the test case declaration to the group description level. There are three ways to do this:

- Linking a test case or script to a test plan using the **Testplan Detail** dialog box to automate the process.
- Link to a test plan manually.
- Record the test case from within the test plan.

Linking to a Test Plan Manually

If you feel comfortable with the syntax of the test plan editor statements and know the locations of the appropriate script and test case, you can enter the `script` and `testcase` statements manually.

1. Place the insertion cursor at the end of a test or group description and press **Enter** to create a new line.
2. Indent the new line one level.
3. Enter the script and/or test case statements using the following syntax:

```
script:
scriptfilename.t testcase:
testcasename
```

Where `script` and `testcase` are keywords followed by a colon, `scriptfilename.t` is the name of the script file, and `testcasename` is the name of the test case.

If you enter a statement correctly, it displays in dark red, the default color used for statements. If not, it will either display in blue, indicating the line is being interpreted as a test description, or black, indicating it is being interpreted as a group description.

Linking a Test Case or Script to a Test Plan using the Testplan Detail Dialog Box

The **Testplan Detail** dialog box automates the process of linking to scripts and test cases. It lets you browse directories and select script and test case names, and it enters the correct the test plan editor syntax into the plan for you.

Linking the Test Plan to Scripts and Test Cases

After you create your test plan, you can associate the appropriate 4Test scripts and test cases that implement your test plan. You create this association by inserting `script` and `testcase` statements in the appropriate locations in the test plan.

There are three ways to link a script or test case to a test plan:

- Linking a description to a script or test case using the **Testplan Detail** dialog box if you want to automate the process of linking scripts and test cases to the test plan.
- Linking to a test plan manually.
- Linking scripts and test cases to a test plan: the test plan editor automatically inserts the `script` and `testcase` statements into the plan once the recording is finished, linking the plan to the 4Test code.

You can insert a `script` and `testcase` statement for each test description, although placing a statement at the group level when possible eliminates redundancy in the test plan. For example, since it is usually good practice to place all the test cases for a given application feature into a single script file, you can

reduce the redundancy in the test plan by specifying the `script` statement at the group level that describes that feature.

You can also insert a `testcase` statement at the group level, although doing so is only appropriate when the test case is data driven, meaning that it receives test data from the plan. Otherwise the same test case would be called several times with no difference in outcome.

Example of Linking a Test Plan to a Test Case

For example, consider the data-driven test case `FindTest`, which takes a record of type `SEARCHINFO` as a parameter:

```
type SEARCHINFO is record
  STRING  sText      // Text to type in document window
  STRING  sPos       // Starting position of search
  STRING  sPattern   // String to look for
  BOOLEAN bCase      // Case-sensitive or not
  STRING  sDirection // Direction of search
  STRING  sExpected  // The expected match

testcase FindTest (SEARCHINFO Data)
  TextEditor.File.New.Pick ()
  DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
  TextEditor.Search.Find.Pick ()
  Find.FindWhat.SetText (Data.sPattern)
  Find.CaseSensitive.SetState (Data.bCase)
  Find.Direction.Select (Data.sDirection)
  Find.FindNext.Click ()
  Find.Cancel.Click ()
  DocumentWindow.Document.VerifySelText ({Data.sExpected})
  TextEditor.File.Close.Pick ()
  MessageBox.No.Click ()
```

The following test plan is associated with the `FindTest` test case. The `testcase` statement occurs at the **Find** dialog group description level, so that each of the test descriptions in the group can call the test case and pass a unique set of data to the test case:

```
Testplan FindTest.pln

Find dialog
script: findtest.t
testcase: FindTest
. . . .
```

Categorizing and Marking Test Plans

This section describes how you can work with selected tests in a test plan.

Marking a Test Plan

Marks are temporary denotations that allow you to work with selected tests in a test plan. For example, you might want to run only those tests that exercise a particular area of the application or to report on only the tests that were assigned to a particular QA engineer. To work with selected tests rather than the entire test plan, you denote or **mark** those tests in the test plan.

Marks can be removed at any time, and last only as long as the current work session. You can recognize a marked test case by the black stripe in the margin.

You can mark test cases by:

- Choice** Select the individual test description, group description, or entire plan that you want to mark, and then choosing the appropriate marking command on the **Testplan** menu.
- Query** You can also mark a test plan according to a certain set of characteristics it possesses. This is called marking by query. You build a query based on one or more specific test characteristics; its script file, data, symbols, or attributes, and then mark those tests that match the criteria set up in the query. For example, you might want to mark all tests that live in the `find.t` script and that were created by the developer named Peter. If you name and save the query, you can reapply it in subsequent work sessions without having to rebuild the query or manually remark the tests that you're interested in working with.
- Test failure** After running a test plan, the generated results file might indicate test failures. You can mark these failures in the plan by selecting **Results > Mark Failures in Plan**. You then might fix the errors and re-run the failed tests.

How the Marking Commands Interact

When you apply a mark using the **Mark** command, the new mark is added to existing marks.

When you mark tests through the query marking commands, the test plan editor by default clears all existing marks before running the query. **Mark by Named Query** supports sophisticated query combinations, and it would not make sense to retain previous marks. However, **Mark by Query**, which allows one-time-only queries, lets you override the default behavior and retain existing marks.

To retain existing marks, uncheck the **Unmark All Before Query** check box in the **Mark by Query** dialog box.

Marking One or More Tests

To mark:

- A single test** Place the cursor on the test description and click **Testplan > Mark**.
- A group of related tests** Place the cursor on the group description and click **Testplan > Mark**. The test plan editor marks the group description, its associated statements, and all test descriptions and statements subordinate to the group description.
- Two or more adjacent tests and their subordinate tests** Select the test description of the adjacent tests and click **Testplan > Mark**. The test plan editor marks the test descriptions and statements of each selected test and any subordinate tests.

Printing Marked Tests

1. Click **File > Print**.
2. In the **Print** dialog box, make sure the **Print Marked Only** check box is checked, as well as any other options you want.
3. Click **OK**.

Mark By Query Dialog Box

You can use the **Mark By Query** dialog box to mark a set of tests in a test plan based on a query which is constructed out of a combination of attributes, symbols, the script, the test case, and the test data. This dialog box is only available when a test plan window is active.

To open the **Mark By Query** dialog box click **Testplan > Mark by Query**.

The dialog box includes the following controls:

Control	Description
Query name	<i>Optional:</i> The name of the query. Named queries are stored by default in the <code>testplan.ini</code> file. The <code>.ini</code> file is specified in the Data File for Attributes and Queries field in the General Options dialog box.
Unmark all before query	Check to clear all marks before running the query. Uncheck to retain existing marks. By default this check box is checked and all marks are cleared.
Mark	Click to close the dialog, run the query against the test plan, and mark all tests in the test plan that are returned by the query.

Test Execution tab

Use the **Test Execution** tab to select the script, test case, and test data in the test plan for which the query should search.

Control	Description
Script	Specify the full path of the script for which the query should search. To browse for the correct script, click Scripts .
Testcase	Specify the name of the test case for which the query should search. To browse for the correct test case, click Testcases . To query for test cases in a specific script you must specify the script before you can specify the test case. To build a query that marks only manual tests, type the keyword <code>manual</code> into the Testcase field.
Test data	Specifies the data values for which the query should search. Separate each value with a comma.

Test Attributes tab

Use the **Test Attributes** tab to add attribute values in the test plan to the query.

Control	Description
Category	Select a <code>Category</code> value to add the tests which belong to this category to the query.
Component	Select a <code>Component</code> value to add the tests which have this component to the query.
Developer	Select a <code>Developer</code> value to add the tests which are developed by this developer to the query.

Symbols tab

Use the **Symbols** tab to add symbols in the test plan to the query.



Note: Do not type the dollar sign (\$) prefix before the symbol name. The wildcard characters * (asterisk) and ? (question mark) are supported for partial matches: * is a placeholder for 0 or more characters, and ? is a placeholder for 1 character.

Control	Description
Edit	Select a symbol from the list and then click Edit to modify the symbol in the text box below the list. Click Replace after you have made your modifications.
Remove	Select a symbol from the list and then click Remove to remove the symbol from the query.
Add	Type the symbol definition in the box below the list of symbols, and then click Add . The newly-defined symbol appears in the list and is added to the query.

Mark By Named Query Dialog Box

Use the **Mark By Named Query** dialog box to create, combine, edit, or delete named queries. To use named queries, you must have defined attributes.

Click **Testplan > Mark by Named Query**.

Named queries are stored by default in `testplan.ini`. The `testplan.ini` file is specified in the **Data File for Attributes and Queries** field in the **General Options** dialog box.

The **Mark By Named Query** dialog is only available when a test plan is open and the test plan window is active.

Testplan Queries	Displays a list of existing test plan queries.
Mark	Click to mark the selected query within the test plan.
New	Click to access the New/Edit Testplan Query dialog box where you can create a new test plan query.
Combine	Click to access the Combine Testplan Queries dialog where you can select a query to combine with the currently selected test plan query.
Edit	Click to access the New/Edit Testplan Query dialog box where you can modify the currently selected test plan. The Edit button is not available until you select an existing query.
Remove	Click to delete the selected test plan query. The Remove button is not available until you select an existing query.

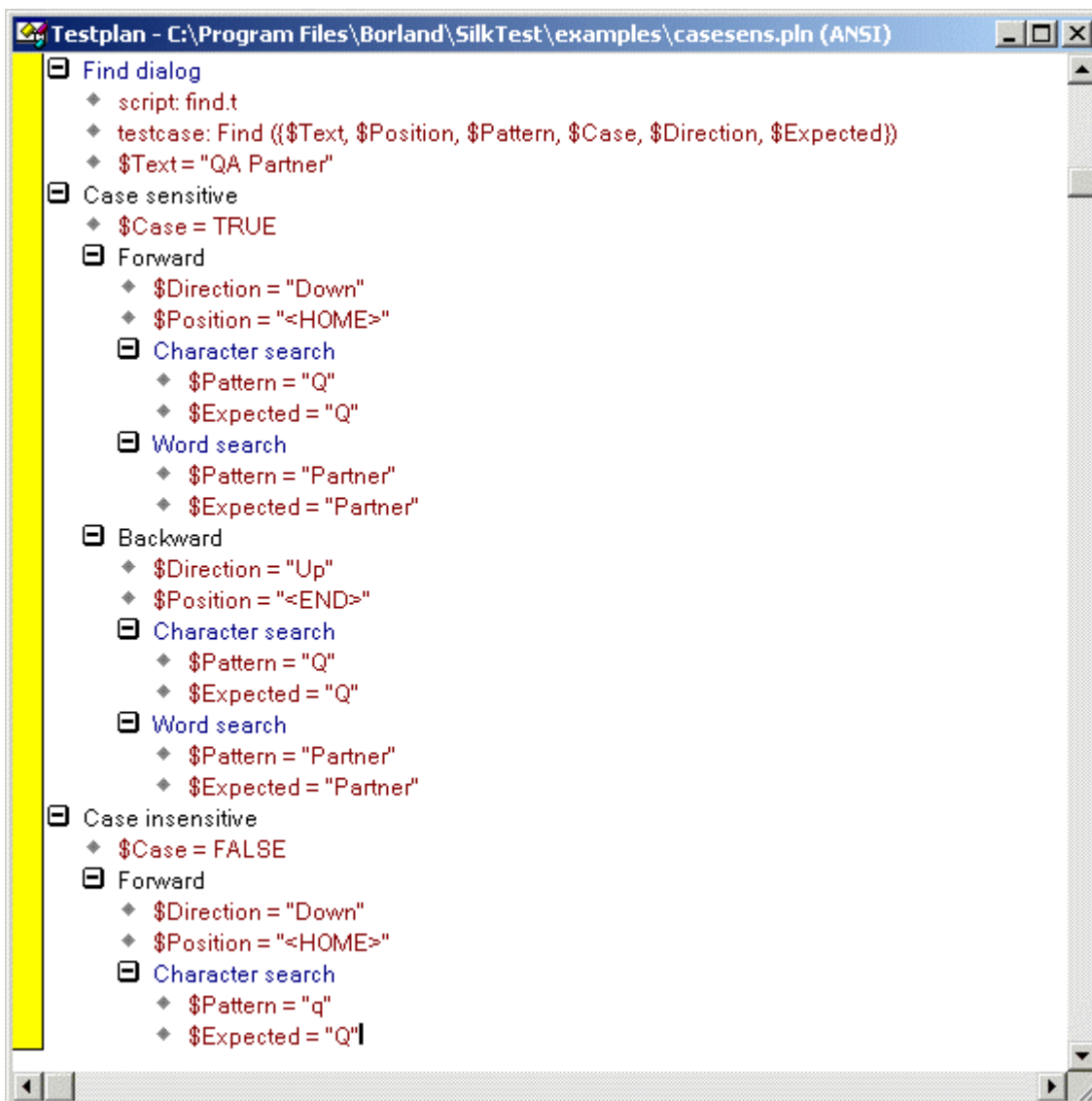
Using Symbols

This section describes symbols, which represent pieces of data in a data driven test case.

Overview of Symbols

A symbol represents a piece of data in a data driven test case. It is like a 4Test identifier, except that its name begins with the `§` character. The value of a symbol can be assigned locally or inherited. Locally assigned symbols display in black and symbols that inherit their value display in blue in the **Testplan Detail** dialog box.

For example, consider the following test plan:



The test plan in the figure uses six symbols:

- `$Text` is the text to enter in the document window.
- `$Position` is the position of the insertion point in the document window.
- `$Pattern` is the pattern to search for in the document window.
- `$Case` is the state of the **Case Sensitive** check box.
- `$Direction` is the direction of the search.
- `$Expected` is the expected match.

The symbols are named in the parameter list to the `FindTest` testcase, within the parentheses after the test case name.

```
testcase: FindTest ({ $Text, $Position, $Pattern, $Case, $Direction,
$Expected })
```

- The symbols are only named in the parameter list; they are not assigned values. The values are assigned at either the group or test description level, depending on whether the values are shared by several tests or are unique to a single test. If a symbol is defined at a level in the plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value it had at the higher level. You can tell whether a symbol is locally assigned by using the **Testplan Detail** dialog box: Locally assigned symbols display in black. Symbols that inherit their values display in blue.

For example, in the preceding figure, each test description assigns its own unique values to the \$Pattern and the \$Expected symbols. The remaining four symbols are assigned values at a group description level:

- The \$Text symbol is assigned its value at the Find dialog group description level, because all eight tests of the Find dialog enter the text Silk Test Classic into the document window of the Text Editor application.
- The \$Case symbol is assigned the value TRUE at the Case sensitive group description level and the value FALSE at the Case insensitive group description level.
- The \$Direction symbol is assigned the value Down at the Forward group description level, and the value Up at the Backward group description level.
- The \$Position symbol is assigned the value <HOME> at the Forward group description level, and the value <END> at the Backward group description level.

Because the data that is common is factored out and defined at a higher level, it is easy to see exactly what is unique to each test.

Symbol Definition Statements in the Test Plan Editor

Use symbols to define data that is shared by a group of tests in the plan. Symbol definitions follow these syntax conventions:

- The symbol name can be any valid 4Test identifier name, but must begin with the \$ character.
- The symbol value can be any text. When the test plan editor encounters the symbol, it expands it (in the same sense that another language expands macros). For example, the following test plan editor statement defines a symbol named Color and assigns it the STRING value "Red":

```
$Color = "Red"
```

- To use a \$ in a symbol value, precede it with another \$. Otherwise, the compiler will interpret everything after the \$ as another symbol. For example, this statement defines a symbol with the value Some \$String: `$MySymbol = "Some$$String"`
- To assign a null value to a symbol, do not specify a value after the equals sign. For example: `$MyNullSymbol =`
- To indicate that a test is incomplete when generating a test plan completion report, assign the symbol the ? character. For example: `$MySymbol = ?`

If a symbol is listed in the argument list of a test case, but is not assigned a value before the test case is actually called, the test plan editor generates a runtime error that indicates that the symbol is undefined. To avoid this error, assign the symbol a value or a ? if the data is not yet finalized.

Defining Symbols in the Testplan Detail Dialog box

Place the insertion cursor in the plan where you need to assign a value to a symbol.

1. Click **Testplan > Detail**.
2. Select the **Symbols** tab on the **Testplan Detail** dialog box, and enter the symbol definition in the text box to the left of the **Add** button.
You do not need to enter the \$ character; the test plan editor takes care of this for you when it inserts the definitions into the test plan.
3. Click **Add**. Silk Test Classic adds the symbol to the list box above the **Add text** text box.
4. Define additional symbols in the same manner, and then click **OK** when finished.

Silk Test Classic closes the **Testplan Detail** dialog box and enters the symbol definitions, including the \$ character, into the plan. If a symbol is defined at a level in the plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value it had at the higher level. You can tell whether a symbol is locally assigned by using the **Testplan Detail** dialog box: Locally assigned symbols display in black. Symbols that inherit their values display in blue.

Assigning a Value to a Symbol

You can define symbols and assign values to them by typing them into the test plan, using this syntax:

```
$symbolname = symbolvalue
```

where `symbolname` is any valid 4Test identifier name, prefixed with the `$` character and `symbolvalue` is any string, list, array, or the `?` character (which indicates an undefined value).

For example, the following statement defines a symbol named `Color` and assigns it the `STRING` value `"Red"`:

```
$Color = "Red"
```

If a symbol is defined at a level in the plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value it had at the higher level.

Specifying Symbols as Arguments when Entering a testcase Statement

1. Place the insertion cursor in the test plan at the location where the `testcase` statement is to be inserted. Placing a symbol name in the argument list of a `testcase` statement only specifies the name of the symbol; you also need to define the symbol and assign it a value at either the group or test case description level, as appropriate.

If you do not know the value when you are initially writing the test plan, assign a question mark (`?`) to avoid getting a compiler error when you compile the test plan; doing so will also cause the tests to be counted as incomplete when a **Completion report** is generated.

2. Click **Testplan > Detail**.
3. Enter the name of a data driven test case on the **Testplan Detail** dialog box, followed by the argument list enclosed in parenthesis. If the test case expects a record, and not individual values, you must use the list constructor operator (curly braces).
4. Click **OK**. Silk Test Classic dismisses the **Testplan Detail** dialog box and inserts the `testcase` statement into the test plan.

Attributes and Values

This section describes site-specific characteristics that you can define for your test plan and assign to test descriptions and group descriptions.

Overview of Attributes and Values

Attributes are site-specific characteristics that you can define for your test plan and assign to test descriptions and group descriptions. Attributes are used to categorize tests, so that you can reference them as a group. Attributes can also be incorporated into queries, which allow you to mark tests that match the query's criteria. Marked tests can be run as a group.

By assigning attributes to parts of the test plan, you can:

- Group tests in the plan to distinguish them from the whole test plan.
- Report on the test plan based on a given attribute value.
- Run parts of the test plan that have a given attribute value.

For example, you might define an attribute called `Engineer` that represents the set of QA engineers that are testing an application through a given test plan. You might then define values for `Engineer` like `David`, `Jesse`, `Craig`, and `Zoe`, the individual engineers who are testing this plan. You can then assign the values of

Engineer to the tests in the test plan. Certain tests are assigned the value of David, others the value of Craig, and so on. You can then run a query to mark the tests that have a given value for the Engineer attribute. Finally, you can run just these marked tests.

Attributes are also used to generate reports. For example, to generate a report on the number of passed and failed tests for Engineer Craig, simply select this value from the **Pass/Fail Report** dialog box. You do not need to mark the tests or build a query in this case.

Attributes and values, as well as queries, are stored by default in `testplan.ini` which is located in the Silk Test Classic installation directory. The initialization file is specified in the **Data File for Attributes and Queries** field in the **General Options** dialog box.

Silk Test Classic ships with predefined attributes. You can also create up to 254 user-defined attributes.

Make sure that all the QA engineers in your group use the same initialization body file. You can modify the definition of an attribute.

Modifying attributes and values through the **Define Attributes** dialog box has no effect on existing attributes and values already assigned to the test plan. You must make the changes in the test plan yourself.

Predefined Attributes

The test plan editor has three predefined attributes:

- Developer** Specifies the group of QA engineers who developed the test cases called by the test plan.
- Component** Specifies the application modules to be tested in this test plan.
- Category** Specifies the kinds of tests used in your QA Department, for example, Smoke Test.

User Defined Attributes

You can define up to 254 attributes. You can also rename the predefined attributes.

The rules for naming attributes include:

- Attribute names can be up to 11 characters long.
- Attribute and value names are not case sensitive.

Adding or Removing Members of a Set Attribute

You can assign multiple values to test attributes of the type `Set`.

For example, you might have a `Set` attribute called `RunWhen` with the following three values:

- `UI`
- `regression`
- `smoke`

You can assign any combination of these three values to a test or a group of tests. Separate each value with a semicolon.

You can use the `+` operator to add members to a `Set` attribute and the `-` operator to remove members from a `Set` attribute.

Example 1: Using `+` to add values

```
RunWhen: UI; regression Test 1 testcase: t1 RunWhen: + smoke
Test 2 testcase: t2
```

In this example, `Test 1` has the values `UI` and `regression`.

The statement `RunWhen: + smoke` adds the value `smoke` to the previously assigned values, so *Test 2* has the values `UI`, `regression`, and `smoke`.

Example 2: Using - to remove values

```
RunWhen: UI; regression Test 1 testcase: t1 RunWhen: -  
regression Test 2 testcase: t2
```

In this example, *Test 1* has the values `UI` and `regression`.

The statement `RunWhen: - regression` removes the value `regression` from the previously assigned values, so *Test 2* has the value `UI`.

Rules for Using + and -

- You must follow the + or - with a space.
- You can add or remove any number of elements with one statement. Separate each element with a semicolon.
- You can specify + elements even if no assignments had previously been made. The result is that the elements are now assigned.
- You can specify - elements even if no assignments had previously been made. The result is that the set's complement is assigned. Using the previous example, specifying:

```
RunWhen: - regression
```

when no `RunWhen` assignment had previously been made results in the values `UI` and `smoke` being assigned.

Defining an Attribute and the Values of the Attribute

1. Click **Testplan > Define Attributes**, and then click **New**.
2. Type a name for the new attribute into the **Name** field.
3. Select one of the following types, and then click **OK**.

Normal You specify values when you define the attribute. Users of the attribute in a test plan pick one value from the list.

Edit You don't specify values when you define the attribute. Users type their own values when they use the attribute in a test plan.

Set Like normal, except that users can pick more than one value.

4. On the **Define Attributes** dialog box, if you:

- have defined an `Edit` type attribute, you are done. Click **OK** to close the dialog box.
- are defining a `Normal` or `Set` type attribute, type a value into the **Values** field and click **Add**.

Once attributes have been defined, you can modify them.

Assigning Attributes and Values to a Test Plan

Attributes and values have no connection to a test plan until you assign them to one or more tests using an assignment statement. To add an assignment statement, you can do one of the following:

- Type the assignment statement yourself directly in the test plan.
- Use the **Testplan Detail** dialog box.

Format

An assignment statement consists of the attribute name, a colon, and a valid attribute value, in this format:

```
attribute-name: attribute value
```

For example, the assignment statement that associates the *Searching* value of the *Module* attribute to a given test would look like:

```
Module: Searching
```

Attributes of type *Set* are represented in this format:

```
attribute-name: attribute value; attribute  
value; attribute value; ...
```

Placement

Whether you type an assignment statement yourself or have the **Testplan Detail** dialog box enter it for you, the position of the statement in the plan is important.

To have an assignment statement apply to	Place it directly after the
An individual test	test description
A group of tests	group description

Assigning an Attribute from the Testplan Detail Dialog Box

1. Place the cursor in the test plan where you would like the assignment statement to display, either after the test description or the group description.
2. Click **Testplan > Detail**, and then click the **Test Attributes** tab on the **Testplan Detail** dialog box. The arrow in the list box at the top of the dialog box identifies the test description at the cursor position in the test plan. The attribute will be added to this test description. The **Test Attributes** tab lists all your current attributes at this level of the test plan.
3. Do one of the following:
 - If the attribute is of type *Normal*, select a value from the list.
 - If the attribute is of type *Set*, select one or more values from the list.
 - If the attribute is of type *Edit*, type a value.
4. Click **OK**. Silk Test Classic closes the dialog box and places the assignment statements in the test plan.

Modifying the Definition of an Attribute

Be aware that modifying attributes and values through the **Define Attributes** dialog box has no effect on existing attributes and values already assigned to the test plan. You must make the changes in the test plan yourself.

1. Click **Testplan > Define Attributes**.
2. On the **Define Attributes** dialog box, select the attribute you want to modify, then:

Rename an attribute Edit the name in the **Name** text box.

Assign a new value to the attribute Type the value in the text box at the bottom right of the dialog box, and click **Add**. The value is added to the list of values.

Modify a value Select the value from the **Values** list box, and click **Edit**. The value displays in the text box at the bottom right of the dialog box and the **Add** button is renamed to **Replace**. Modify the value and click **Replace**.

Delete a value Select the value from the **Values** list box and click **Remove**. The text box is cleared and the value is removed from the **Values** list box.

Delete an attribute Click **Delete**.

3. Click **OK**. The attributes and values are saved in the initialization file specified in the **General Options** dialog box.

Queries

This section describes how you can use a test plan query to mark all tests that match a user-selected set of criteria, or test characteristics.

Overview of Test Plan Queries

You can use a test plan query to mark all tests that match a user-selected set of criteria, or test characteristics. A query comprises one or more of the following criteria:

- Test plan execution: script file, test case name, or test data
- Test attributes and values
- Symbols and values

Test attributes and symbols must have been previously defined to be used in a query.

Named queries are stored by default in `testplan.ini`. The initialization file is specified in the **Data File for Attributes and Queries** text box in the **General Options** dialog box. The `testplan.ini` file is in the Silk Test Classic installation directory. Make sure that all the QA engineers in your group use the same initialization file.

Overview of Combining Queries to Create a New Query

You can combine two or more existing queries into a new query using the **Mark by Named Query** dialog box. The new query can represent the union of the constituent queries (logical OR) or the intersection of the constituent queries (logical AND).

Combining by union

Combining two or more queries by union creates a new named query that marks all tests that would have been marked by running each query one after the other while retaining existing marks. Since **Mark by Named Query** clears existing marks before running a query, the only way to achieve this result is to create a new query that combines the constituent queries by union.

Example

Suppose you have two queries, Query1 and Query2, that you want to combine by union.

Query1	Query2
Developer: David	Developer: Jesse
Component: Searching	TestLevel: 2

The new query created from the union of Query1 and Query2 will first mark those tests that match all the criteria in Query1 (Developer is David and Component is Searching) and then mark those tests that match all the criteria in Query2 (Developer is Jesse and TestLevel is 2).

Combining by intersection

Combining two or more queries by intersection creates a new named query that marks every test that has the criteria specified in all constituent queries.

Example

For example, combining Query1 and Query2 by intersection would create a new query that comprised these criteria: Developer is David and Jesse, Component is Searching, and TestLevel is 2. In this case, the new query would not mark any tests, since it is impossible for a test to have two different values for the attribute Developer (unless Developer were defined as type Set under Windows). Use care when combining queries by intersection.

Guidelines for Including Symbols in a Query

- Use ? (question mark) to indicate an unset value. For example, `Mysymbol = ?` in a query would mark those tests where *Mysymbol* is unset. Space around the equals sign (=) is insignificant.
- If you need to modify the symbol in the query, select it from the list box and click **Edit**. The test plan editor places it in the text box and changes the **Add** button to **Replace**. Edit the symbol or value and click **Replace**.
- To exclude the symbol from the query, select it from the list box and click **Remove**. The test plan editor deletes it from the list box.

The Differences between Query and Named Query Commands

Testplan > Mark by Query or **Testplan > Mark by Named Query** both create queries, however, **Mark by Named Query** provides extra features, like the ability to combine queries or to create a query without running it immediately. If the query-creation function and the query-running function are distinct in your company, then use **Mark by Named Query**. If you intend to run a query only once, or run a query while keeping existing marks, then use **Mark by Query**.

The following table highlights the differences between the two commands.

Mark by Query	Mark by Named Query
Builds a query based on criteria you select and runs query immediately.	Builds a new query based on criteria you select. Can run query at any time.
Name is optional, but note that only named queries are saved and can be rerun at any time in the Mark by Named Query dialog box.	Name is required. Query is saved.
Cannot edit or delete a query.	Can edit or delete a query.
Cannot combine queries.	Can combine queries into a new query.
Lets you decide whether or not to clear existing marks before running new query. Unmarks by default.	Clears existing marks before running new query.

Unnamed queries can be run only once. If you name the query, you can have the test plan editor run it in the same or subsequent work sessions without having to rebuild the query or manually remark the tests that you're interested in rerunning or reporting on.

Creating a New Query

You can create a new query for a test plan through either **Testplan > Mark by Query** or **Testplan > Mark by Named Query**. You can also create a new query by combining existing queries.

1. Open the test plan and any associated sub-plans.
2. Click **Testplan > Mark by Query** or **Testplan > Mark by Named Query**.
3. Identify the criteria you want to include in the query.
 - To include a script, a test case, or test data, use the **Test Execution** tab. Click **Script** to select a script and click **Testcase** to select a test case, or type the full specification yourself. To build a query that marks only manual tests, type the keyword into the **Testccase** field.
 - To include existing attributes and values in the query, use the **Test Attributes** tab.
 - To include one or more existing symbols and values, use the **Symbols** tab. Type the information into the **Add** field and click **Add**. The symbol and value are added to the list box.

Do not type the dollar sign (\$) prefix before the symbol name. The wildcard characters * (asterisk) and ? (question mark) are supported for partial matches: * is a placeholder for 0 or more characters, and ? is a placeholder for 1 character.

Example 1

If you type `find_5` (* in the **Testcase** field, the query searches all the `testcase` statements in the plan and marks those test descriptions that match, as well as all subordinate descriptions to which the matching `testcase` statement applies, which are the test cases where the `find_5` test case passed in data.

Example 2

If you type `find.t` in the **Script** field, the query searches all `script` statements in the plan and marks those test descriptions that match exactly, as well as all subordinate descriptions to which the matching `script` statement applies, which are the test cases in which you had specified `find.t` exactly. It would not match any `script` statements in which you had specified a full path.

4. Take one of the following actions, depending on the command you chose to create the query:

Mark by Query Click **Mark** to run the query against the test plan. The test plan editor closes the dialog box and marks the test plan, retaining the existing marks if requested.

Mark by Named Query Click **OK** to create the query. The **New Testplan Query** dialog box closes, and the **Mark by Named Query** dialog box is once again visible. The new query displays in the **Testplan Queries** list box.

If you want to:

- Run the query, select it from the list box and click **Mark**.
- Close the dialog box without running the query, click **Close**.

Edit a Query

1. Click **Testplan > Mark by Named Query** to display the **Mark by Named Query** dialog box.
2. Select a query from the **Testplan Queries** list box and click **Edit**.
3. On the **Edit Testplan Query** dialog box, edit the information as appropriate, and then click **OK**.
4. To run the query you just edited, select the query and click **Mark**. To close the dialog box without running the edited query, click **Close**.

Delete a Query

1. Click **Testplan > Mark by Named Query** to open the **Mark by Named Query** dialog box.
2. Select a query from the **Testplan Queries** box and click **Remove**.
3. Click **Yes** to delete the query, and then click **Close** to close the dialog box.

Combining Queries

1. Click **Testplan > Mark by Named Query** to display the **Mark by Named Query** dialog box.
2. Click **Combine**. The **Combine Testplan Queries** dialog box lists all existing named queries in the **Queries to Combine** list box.
3. Specify a name for the new query in the **Query Name** text box.
4. Select two or more queries to combine from the **Queries to Combine** list box.
5. Click the option button that represents the combination method to use: either **Union of Queries** or **Intersection of Queries**.
6. Click **OK** to save the new query. The **Mark by Named Query** dialog box displays with the new query in the **Testplan Queries** list box.
7. To run the query, select the query and click **Mark** or click **Close** to close the dialog box without running the query.

Combine Testplan Queries Dialog Box

Use to create a new query by combining existing queries.

Click **Testplan > Mark by Named Query** and then click **Combine**.

Query name	Type the name of the new query that you want to create by combining existing queries.
Queries to combine	Displays a list of named queries. Select two or more queries from the list to combine.
Union of queries	Click to combine two or more queries by union. This option creates a new named query that marks all tests that would have been marked by running each query, one after the other, while retaining existing marks. Since Mark by Named Query clears existing marks before running a query, the only way to achieve this result is to create a new query that combines the queries by union.
Intersection of queries	Click to combine two or more queries by intersection. This option creates a new named query that marks every test that has the criteria specified in all constituent queries. Use care when combining queries by intersection.

New/Edit Testplan Query Dialog Box

Use to construct a query from attributes, symbols, the script, the test case, and test data. This dialog is only available when a test plan is open and the test plan window is active.

Click **Testplan > Mark by Named Query** and then click **New** or **Edit**. To access the **Edit Testplan Query** dialog box, you must select an existing test plan query.

Query name	Type the name of the query. This is optional. Named queries are stored by default in <code>testplan.ini</code> . The initialization file is specified in the Data File for Attributes and Queries field in the General Options dialog box.
-------------------	--

Test Execution tab

Script	Specifies the full path of the script that you want to display on the current line of the test plan. To browse for the correct script, click Scripts to display the Testplan Detail Scripts dialog
---------------	--

box. Once you select a script, the **Testplan Editor** writes the selected script into the test plan, beginning the line with the reserved word `script`, followed by a colon.

Test case Specifies the name of the test case that you want to display on the current line of the test plan. To browse for the correct test case, click **Test Cases** to view the test cases contained in the script that you have specified in the **Script** field. Once you select a test plan from the **Testplan Detail Testcase** dialog box, the **Testplan Editor** writes the selected test case into the test plan, beginning the line with the reserved word `testcase`, `ddatestcase` for a data-driven test case, followed by a colon. To build a query that marks only manual tests, type the keyword `manual` into the **Test case** field.

Test data Specifies the data values that you want to pass to the test case. Separate each value with a comma. The **Testplan Editor** writes the data you specify here to the current line of the test plan, beginning the line with the reserved word `testdata`, `ddatestdata` for a data-driven test case, followed by a colon.

Scripts Click to select a script to include in the query or type the script name into the **Script** field.

Test Cases Click to select a test case to include in the query or type the test case name into the **Test case** field.

Test Attributes tab

Use the **Test Attributes** tab to assign existing attribute values to your test plan. For information about adding new attributes and values, see *Defining an Attribute and the Values of the Attribute*.

Category Lists the available categories. Click the arrow to select a category from the list.

Component Lists the available components. Click the arrow to select a component from the list.

Developer Displays the available developers. Click the arrow to select a developer from the list.

Symbols tab



Note: Do not type the dollar sign (\$) prefix before the symbol name. The wildcard characters * (asterisk) and ? (question mark) are supported for partial matches: * is a placeholder for 0 or more characters, and ? is a placeholder for 1 character.

Symbols will be ignored for a data-driven test case.

Edit Select a symbol from the list and then click **Edit** to modify the symbol in the field below the symbols list. Click **Replace** after you have made your modifications.

Remove Select a symbol from the list and then click **Remove** to remove the symbol from the test plan.

Add Type the symbol definition in the field below the symbols list, and then click **Add**. The newly-defined symbol displays in the list. You do not need to enter the \$ character because the **Testplan Editor** takes care of this when it inserts the definitions into the test plan.

Create Session Dialog Box

You can use the **Create Session** dialog box to create a new manual test session. Select a product, then use the list boxes to enter test data such as component, platform, version, and build. You can use the **Named Query** box to select tests that have been marked by a Named Query.

When you are using the **Create Session** dialog box for a new manual session, inactive and active products are displayed.

In order for reports to properly reflect your manual session, it is important to completely assign test data. For example, if you do not assign a build to this session, this session is not reflected on any reports that are based on builds.

Testplan Detail Dialog Box

Use the **Testplan Detail** dialog box to:

- Specify valid script, test case, and test data statements.
- Assign attribute values to your test plan.
- Add, edit, or delete a symbol associated with the test plan.

This dialog box is only available when a test plan is open and the **Testplan** dialog box is active.

Click **Testplan > Testplan Detail**. You can also place your cursor in a test case, click the right mouse button, and then select **Test Details**.

The box at the top of the dialog displays your cursor position in the test plan when you opened the dialog box. The black arrow in the left margin identifies the test at which your cursor is positioned in the test plan. This is the test description to which you are adding details. Any details will apply to this test and any tests that inherit details from this test. If the test case and script associated with the current test description are inherited from a higher level in the test plan, they are shown in blue; otherwise, they are shown in black.

Test Execution tab

- Script** Specifies the full path of the script that you want to appear on the current line of the test plan. To browse for the correct script, click **Scripts** to display the **Testplan Detail - Script** dialog. Once you select a script, the **Testplan Editor** writes the selected script into the test plan, beginning the line with the reserved word `script`, followed by a colon.
- Test case** Specifies the name of the test case that you want to display on the current line of the test plan. To browse for the correct test case, click **Test Cases** to view the test cases contained in the script you have specified in the **Script** field. Once you select a test plan from the **Testplan Detail - Testcase** dialog box, the **Testplan Editor** writes the selected test case into the test plan, beginning the line with the reserved word `testcase`, or `ddatestcase` for a data-driven test case, followed by a colon. Selecting the test case ensures that the proper keywords are inserted into the test plan.
- Test data** Specifies the data values that you want to pass to the test case. Separate each value with a comma. The **Testplan Editor** writes the data you specify here to the current line of the test plan, beginning the line with the reserved word `testdata`, or `ddatestcase` for a data-driven test case, followed by a colon. To select certain rows to use to run the data-driven test case, click **Specify Rows** to view the **Specify Rows** dialog box.

Test Attributes tab

Use the **Test Attributes** tab to assign existing attribute values to your test plan.

- Category** Displays the categories available. Click the arrow to select from the list of available categories.
- Component** Displays the components available. Click the arrow to select from the list of available components.
- Developer** Displays the available developers. Click the arrow to select from the list of available developers.

Symbols tab

Use the **Symbols** tab to add, edit, or delete a symbol associated with the test. If a symbol is defined at a level in the test plan where it can be shared by a group of tests, each test can assign its own local value to the symbol, overriding whatever value the symbol had at the higher level. Locally assigned symbols display in black. Symbols that inherit their values display in blue.



Note: Do not type the dollar sign (\$) prefix before the symbol name. The wildcard characters * (asterisk) and ? (question mark) are supported for partial matches: * is a placeholder for 0 or more characters, and ? is a placeholder for 1 character. Symbols are ignored for a data-driven test case.

Edit Select a symbol from the list and then click **Edit** to modify the symbol in the text box below the list. Click **Replace** after you have made your modifications.

Remove Select a symbol from the list and then click **Remove** to remove the symbol from the test plan.

Add Type the symbol definition in the box below the list of symbols and then click **Add**. The newly-defined symbol displays in the list. You do not need to enter the \$ character because the **Testplan Editor** takes care of this for you when it inserts the definitions into the test plan.

Testplan Detail - Testcase Dialog Box

Use the **Testplan Detail - Testcase** dialog box to select a test case to add to the currently selected test plan.

Click **Testplan > Detail**, click the **Test Execution** tab, specify a script in the **Script** field, and then click **Testcases**.

Testcase Lists the test cases available in the currently selected script. Select the test case that you want to enter into the test plan from the list. Silk Test Classic enters the correct **Testplan Editor** syntax into the test plan.

Define Attributes Dialog Box

Use to define or rename the **Testplan Editor** attributes, as well as add, rename, or remove attribute values. This dialog box is only available when a test plan is open and the test plan window is active. Attribute values are stored in the `testplan.ini` file.

To open the **Define Attributes** dialog box, click **Testplan > Define Attributes** in the Silk Test Classic menu.

Attributes Displays the list of attributes defined in the **Testplan Editor**. An attribute is a site-specific characteristic that you define for your test plan and associate with tests. Attributes allow you to query or execute sections of the test plan or report on various characteristics. The **Testplan Editor** has three predefined attributes:

Category Specifies the type of test case or the type of the group of test cases used in your QA Department, for example *Smoke Test*.

Component Specifies the pages or application modules of the application under test which should be tested in this test plan.

Developer Specifies the group of QA engineers which are assigned to develop the test case or group of test cases in the test plan.

New	Displays the New Attribute dialog box, which you can use to define a new attribute. You can then use the Define Attributes dialog box to assign values to the new attribute.
Delete	Deletes the selected attribute. When you delete an attribute, you also delete the values of the attribute.
Name	Displays the name of the attribute that you have selected in the Attributes list. You can edit the attribute name by modifying the text in the Name field.
Type	Displays the type that is associated with the currently selected attribute in the Attributes list. An attribute may be one of the following types: <ul style="list-style-type: none"> • <i>Normal</i> • <i>Edit</i> • <i>Set</i> <p>This field is display-only.</p>
Values	Displays the values that are associated with the selected attribute type.
Edit	Click to modify the selected attribute name and value. You must select an attribute from the Attributes list and a value from the Values list before the Edit button is available. To modify the attribute name, edit the text in the Name field. To modify the value, edit the text in the field next to the Replace button, and then click Replace .
Remove	Removes the selected value from the list. You must select an attribute from the Attributes list and a value from the Values list before the Remove button is available. To remove an attribute, you must select the attribute from the Attributes list, and then click Delete .
Add	Allows you to add a new value that should be associated with the selected attribute type. You must select an attribute from the Attributes list and type a value in the text field before the Add button is available. You cannot assign values for attributes of type <i>Edit</i> . Attribute values are stored in the <code>testplan.ini</code> file.

New Attribute Dialog Box

Use to define new attributes. You can define up to 254 attributes or rename the predefined attributes, *Developer*, *Component*, and *Category*. This dialog is only available from the **Define Attributes** dialog.

Click **Testplan > Define Attributes** and then click **New**.

Name	Name of the attribute you are defining, using up to 11 characters. Attribute names cannot have spaces and are not case sensitive.
Type	Click to specify a Type for the new attribute you are defining. Specifying a Type is optional. You can select from the following types: <ul style="list-style-type: none"> Normal Values are assigned when defining the attribute. You can pick a value from the list when associating attributes with the testplan statement. Edit Values are not assigned when defining the attribute. You must type your own values when you use the attribute in a testplan. Set Values are assigned when defining the attribute. The Set type is similar to the Normal type, except that you can pick multiple values from the list when associating attributes with the testplan statement.

Update Manual Tests Dialog Box

Use the **Update Manual Tests** dialog box to describe the state of manual tests included in the test plan. This dialog is only available when a test plan that contains manual tests is active.

To open the **Update Manual Tests** dialog box, click **Testplan > Run Manual Tests** .

Plan file	Displays the path and name of the currently selected test plan.
Manual tests	Lists all manual tests contained within the currently selected test plan. Select the manual test that you want to update from this list.
Incomplete	Click to mark the currently selected manual test case incomplete.
Complete	Click to mark the currently selected manual test case complete. This means that a test has been defined. A manual test marked as <i>Complete</i> is tabulated as complete in the Testplan Completion reports.
Has been run	Click to mark the currently selected test case as having been run. The current date and time display in the Run at box. You can then indicate if the test case passed or failed by clicking the appropriate button. A test marked <i>Has been run</i> is considered complete in Testplan Completion reports.
Failed	Click to indicate that the currently selected manual test failed. This button is available only when you select Has been run in the Completion area. Manual tests marked as <i>Failed</i> will be tabulated as such in Pass/Fail reports, as long as you specify the time at which the tests were run in the Run at box.
Passed	Click to indicate that the currently selected manual test passed. This button is available only when you select Has been run in the Completion area. Manual tests marked as <i>Passed</i> will be tabulated as such in Pass/Fail reports, as long as you specify the time at which the tests were run in the Run at box.
Comments for <currently selected manual testcase>	Type any comments you want to associate with the currently selected manual test.
Print	Click to print status information for the manual tests included in the currently selected test plan. This includes the test case name, status, and run information, if applicable.
Run at	Specify when the test was run using this syntax: <code>year-month-day hours:minutes:seconds YYYY-MM-DD HH:MM:SS</code> Hours, minutes, and seconds are optional. For example, enter <code>2011-10-01</code> to indicate the test was run Oct 1, 2011.
Machine	<i>Optional:</i> Specify the name of the machine on which the test was run.

Chart Settings Dialog Box

Use to modify the format in which test plan completion charts appear. This dialog box is only available when a test plan is open and the **Testplan** dialog is active.

Click **Testplan > Completion Report/Setup**.

Chart type	Displays the currently selected chart type:
-------------------	---

- **Bar chart**
- **Line chart**
- **Area chart**

When you select an option, a model is displayed that represents how the chart will look based on current settings.

- Stacked** Check if you want the chart to appear stacked. This is particularly useful when you select a bar chart.
- 3D** Check if you want the chart to appear in three-dimensional format.
- Show Legend** Check if you want to display a legend that describes the data being charted along with your chart.
- Apply** Click to apply the chart settings you selected to your chart.

Duplicate Test Descriptions dialog box

This dialog box is displayed when a managed test plan is saved and duplicate test descriptions are found. These duplicates are usually a result of the editing process.

You can select a test description that will retain the original test ID, or you can use **Reset All IDs** to generate new identifiers for all test descriptions. **Reset All IDs** has database ramifications and affects any existing Silk Central and Issue Manager references to the test description ID. This means that an existing reference will no longer be valid.

Designing and Recording Test Cases with the Open Agent

This section describes how you can design and record test cases with the Open Agent.

Creating Test Cases with the Open Agent

This section describes how you can use the Open Agent to create test cases.

Recording Test Cases for Standard and Web Applications



Note: Before you can record a test case, you have to configure the application that you want to test.

This functionality is supported only if you are using the Open Agent.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. With this approach, you combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.

1. Click **Record Testcase** on the basic workflow bar. If the workflow bar is not visible, click **Workflows > Basic** to enable it. The **Select Test Type** dialog box opens.
2. Select **4Test Testcase**.
3. Click **OK**. The **Record Testcase** dialog box opens.
4. Type the name of your test case into the **Testcase name** field.

Test case names are not case sensitive; they can have any length and they can consist of any combination of alphabetic characters, numerals, and underscore characters.
5. From the **Application State** list box, select **DefaultBaseState** to have the built-in recovery system restore the default base state before the test case begins executing.
 - If you choose *DefaultBaseState* as the application state, the test case is recorded in the script file as `testcase testcase_name ()`.
 - If you choose another application state, the test case is recorded as `testcase testcase_name () appstate appstate_name`.
6. Click **Start Recording**. Silk Test Classic performs the following actions:
 - Closes the **Record Testcase** dialog box.
 - Starts your application, if it was not already running. If you have not configured the application yet, the **Select Application** dialog box opens and you can select the application that you want to test.
 - Removes the editor window from the display.
 - Displays the **Recording** window.
 - Waits for you to take further action.
7. In the application under test, perform the actions that you want to test.

For information about the actions available during recording, see *Actions Available During Recording*.
8. To stop recording, click **Stop** in the **Recorder**. Silk Test Classic displays the **Record Testcase** dialog box, which contains the code that has been recorded for you.

9. To resume recording your interactions, click **Resume Recording**.
10. To add the recorded interactions to a script, click **Paste to Editor** in the **Record Testcase** window. If you have interacted with objects in your application that have not been identified in your include files, the **Update Files** dialog box opens.
11. Perform one of the following steps:
 - Click **Paste testcase and update window declaration(s)** and then click **OK**. In most cases, you want to choose this option.
 - Choose **Paste testcase only** and then click **OK**. This option does not update the window declarations in the INC file when it pastes the script to the Editor. If you previously recorded the window declarations related to this test case, choose this option.
12. Click **OK**. Silk Test Classic adds the recorded actions to the specified script file, and opens the file in the editor.

Recording Test Cases for Mobile Applications

This functionality is supported only if you are using the Open Agent.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. With this approach, you combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.

1. Click **Record Testcase** on the basic workflow bar. If the workflow bar is not visible, click **Workflows > Basic** to enable it. The **Select Test Type** dialog box opens.
2. Select **4Test Testcase**.
3. Click **OK**. The **Record Testcase** dialog box opens.
4. Type the name of your test case into the **Testcase name** field.

Test case names are not case sensitive; they can have any length and they can consist of any combination of alphabetic characters, numerals, and underscore characters.
5. From the **Application State** list box, select **DefaultBaseState** to have the built-in recovery system restore the default base state before the test case begins executing.
 - If you choose *DefaultBaseState* as the application state, the test case is recorded in the script file as `testcase testcase_name ()`.
 - If you choose another application state, the test case is recorded as `testcase testcase_name () appstate appstate_name`.
6. Click **Start Recording**. Silk Test Classic performs the following actions:
 - Closes the **Record Testcase** dialog box.
 - Starts your application, if it was not already running. If you have not configured the application yet, the **Select Application** dialog box opens and you can select the application that you want to test.
 - Removes the editor window from the display.
 - Displays the **Recording** window.
 - Waits for you to take further action.
7. On the device or the Emulator, interact with your native mobile application, driving it to the state that you want to test.
8. In the **Recording** window, perform the actions that you want to record.
 - a) Click on the object with which you want to interact. Silk Test Classic performs the default action for the object. If there is no default action, or if you have to insert text or specify parameters, the **Choose Action** dialog box opens.
 - b) *Optional*: To choose an action for an object, which might not be the default action, right-click on the object. The **Choose Action** dialog box opens.

- c) *Optional*: If the action has parameters, type the parameters into the parameter fields. Silk Test Classic automatically validates the parameters.
- d) Click **OK** to close the **Choose Action** dialog box. Silk Test Classic adds the action to the recorded actions and replays it on the mobile device or emulator.

For information about how to record an interaction with a mobile device, see *Interacting with a Mobile Device*.

9. To verify an image or a property of a control during recording, click `Ctrl+Alt`.

For additional information, see [Adding a Verification to a Script while Recording](#).

10. *Optional*: To interact with an object that is currently not visible in the **Recording** window, use the **Hierarchy View**:

- a) Click **Toggle Hierarchy View**. The **Hierarchy View** opens.
- b) In the object tree, right-click on the object on which you want to perform an action.
- c) Click **Add New Action**. The **Choose Action** dialog box opens.
- d) Proceed as with any other action.

For example, to open the main menu of the device or emulator, right-click on the *MobileDevice* object in the object tree and select the action `PressMenu()`.

11. To pause the recording of interactions with the application, for example to move the application into a different state, click **Pause**.

12. To resume recording interactions, click **Start**.

13. To add the recorded interactions to a script, click **Stop**. If you have interacted with objects in your application that have not been identified in your include files, the **Update Files** dialog box opens.

14. Perform one of the following steps:

- Click **Paste testcase and update window declaration(s)** and then click **OK**. In most cases, you want to choose this option.
- Choose **Paste testcase only** and then click **OK**. This option does not update the window declarations in the INC file when it pastes the script to the Editor. If you previously recorded the window declarations related to this test case, choose this option.

Recording a Test on Microsoft Edge

This functionality is supported only if you are using the Open Agent.

Before you can record a Silk Test Classic test, you must have created a Silk Test Classic project.

When starting the interaction with a web application on Microsoft Edge, Silk Test Classic closes any open instance of Microsoft Edge and starts a new browser. This new browser uses a temporary profile without add-ons and with an empty cache. This instance of Microsoft Edge is closed when shutting down the Open Agent or when starting to test another application outside Microsoft Edge.



Note: You can currently not record keyword-driven tests on Microsoft Edge.

To record a new test for a web application on Microsoft Edge:

1. Select the project to which you want to add the new test.
2. In the toolbar, click **Record Testcase**.
3. Type a name for the new test into the **Testcase name** field.
4. Click **Start Recording**.
5. The **Interactive Recording** window opens and displays the web application. Perform the actions that you want to record.
 - a) Click on the object with which you want to interact. Silk Test Classic performs the default action for the object. If there is no default action, or if you have to insert text or specify parameters, the **Choose Action** dialog box opens.

- b) *Optional:* To choose an action for an object, which might not be the default action, right-click on the object. The **Choose Action** dialog box opens.
 - c) *Optional:* If the action has parameters, type the parameters into the parameter fields. Silk Test Classic automatically validates the parameters.
 - d) Click **OK** to close the **Choose Action** dialog box. Silk Test Classic adds the action to the recorded actions and replays it on the mobile device or emulator.
6. Click **Stop**. The **Record Testcase** dialog box opens.
 7. Perform one of the following steps:
 - Click **Paste testcase and update window declaration(s)** and then click **OK**. In most cases, you want to choose this option.
 - Choose **Paste testcase only** and then click **OK**. This option does not update the window declarations in the INC file when it pastes the script to the Editor. If you previously recorded the window declarations related to this test case, choose this option.
 8. Click **OK**.

Replay the test to ensure that it works as expected. You can modify the test to make changes if necessary.


Recording Window Declarations that Include Locator Keywords

A window declaration specifies a cross-platform, logical name for a GUI object, called the identifier, and maps the identifier to the object's actual name, called the tag or locator. You can use locator keywords, rather than tags, to create scripts that use dynamic object recognition and window declarations. Or, you can include locators and tags in the same window declaration.

To record window declarations that include locator keywords, you must use the Open Agent.

To record window declarations using the Locator Spy:

1. Configure the application to set up the technology domain and base state that your application requires.
2. Click **Record > Window Locators**. The **Locator Spy** opens.
3. Position the mouse over the object that you want to record and perform one of the following steps:
 - Press **Ctrl+Alt** to capture the object hierarchy with the default Record Break key sequence.
 - Press **Ctrl+Shift** to capture the object hierarchy if you specified the alternative Record Break key sequence on the **General Recording Options** page of the **Recording Options** dialog box.

 **Note:** For SAP applications, you must set **Ctrl+Shift** as the shortcut key combination. To change the default setting, click **Options > Recorder** and then check the **OPT_ALTERNATE_RECORD_BREAK** check box.

 - If you use Picking mode, click the object that you want to record and press the Record Break keys.
4. Click **Stop Recording Locator**.

The **Locator** text box displays the XPath query string for the object on which the mouse rests. The **Locator Details** section lists the hierarchy of objects for the locator that displays in the text box. The hierarchy listed in the **Locator Details** section is what will be included in the INC file.
5. To refine the locator, in the **Locator Details** table, click **Show additional locator attributes**, right-click an object and then choose **Expand subtree**. The objects display and any related attributes display in the **Locator Attribute** table.
6. To replace the hierarchy that you recorded, select the locator that you want to use as the parent in the **Locator Details** table. The new locator displays in the **Locator** text box.
7. Perform one of the following steps:
 - To add the window declarations to the INC file for the project, position your cursor where you want to add the window declarations in the INC file, and then click **Paste Hierarchy to Editor**.

- To copy the window declarations to the Clipboard, click **Copy Hierarchy to Clipboard** and then paste the window declarations into a different editing window or into the current window at the location of your choice.

8. Click **Close**.

Recording Locators Using the Locator Spy

This functionality is supported only if you are using the Open Agent.

Use the *Locator Spy* to record the locator of a specific object in your application under test. You can then copy the locator to the test case or to the Clipboard.

1. Configure the application to set up the technology domain and base state that your application requires.
2. Click **File > New**. The **New File** dialog box opens.
3. Select **4Test script** and then click **OK**. A new 4Test Script window opens.
4. Click **Record > Window Locators**.



Note: If you have not configured the application yet, the **Configure Test** dialog box opens and you can select the application that you want to test.

The **Locator Spy** opens.

5. Position the mouse over the object that you want to record. The related locator XPath query string shows in the **Selected Locator** text box. The **Locator Details** section lists the hierarchy of objects for the locator that displays in the text box.



Note: If you are testing on a browser, the **Selected Locator** field displays the locator only when you actually capture it.

6. Perform one of the following steps:

- Press **Ctrl+Alt** to capture the object. The **Locator** field displays the XPath query string for the object. You can edit the locator.
- Press **Ctrl+Shift** to capture the object if you specified the alternative Record Break key sequence on the **General Recording Options** page of the **Recording Options** dialog box.



Note: For SAP applications, you must set **Ctrl+Shift** as the shortcut key combination to use to pause recording. To change the default setting, click **Options > Recorder** and then check the **OPT_ALTERNATE_RECORD_BREAK** check box.

- Click **Stop Recording Locator** to capture the locator that is currently displayed in the **Locator** field.
- If you use **Picking** mode, click the object that you want to record and press the Record Break keys.



Note: Silk Test Classic does not verify whether the locator string is unique. Micro Focus recommends that you ensure that the string is unique, because otherwise additional objects might be found when you run the test. Furthermore, you might want to exclude some of the attributes that Silk Test Classic identifies because the string will work without them.

7. Click **Validate Locator** to highlight the object, to which the locator in the **Locator** field corresponds, in the test application.
8. To refine the locator, in the **Locator Details** table, click **Show additional locator attributes**, right-click an object and then choose **Expand subtree**. The objects display and any related attributes display in the **Locator Attribute** table.
9. *Optional:* You can replace a recorded locator attribute with another locator attribute from the **Locator Details** table.

For example, your recorded locator might look like the following:

```
/Window[@caption='MyApp']//Control[@id='table1']
```

If you have a caption `Files` listed in the **Locator Details** table, you can manually change the locator to the following:

```
/Window[@caption='MyApp']//Control[@caption='Files']
```

The new locator displays in the **Selected Locator** text box.

10. Copy the locator to the test case or to the Clipboard.

- Click **Paste Hierarchy to Editor** to paste the window declarations that are displayed in the **Locator Details** into the open Silk Test Classic file.
- Click **Copy Hierarchy to Clipboard** to copy the window declarations that are displayed in the **Locator Details** to the Clipboard.
- Click **Paste Locator to Editor** to paste the contents of the **Locator** field into the open Silk Test Classic file.
- Click **Copy Locator to Clipboard** to copy the contents of the **Locator** field to the Clipboard.



Tip: If you have copied code to the Clipboard, you can click **Edit > Paste** in the Silk Test Classic menu to insert the code into the current window at the location of your choice, or even into a different editing window.

11. Click **Close**.

Recording an Application State

You can define an application state routine that Silk Test Classic runs before it executes your test case. You have to create the application state before recording the test cases that are associated with the application state. You can manually write an application state routine or you can use the **Record Application State** dialog box to record an application state.

To record an application state:

1. Open the file in which you want to place the application state.
This can either be the test frame file for the application or the script file where the associated test cases are defined.
If you put the application state in the test frame file, it will be available to all test cases. If you put it in the script file, it will be available only to test cases in that script file.
2. Open the application that you want to test.
3. Click **Record > Application State**. If the current window is not a script or an include file, Silk Test Classic prompts you to create a new include file.
4. Type the name of your new application state in the **Application State Name** text box.
The application state name must be less than 64 characters.
5. Select an application state from the **Based On** list box.
This specifies an existing application state on which the new application state is based. By default, the last application state that you have specified in this field is selected.
6. If you are using the Classic Agent, you can check the **Show AppState status window** check box to display a status window while Silk Test Classic is driving the application to the specified base state. If the status window obscures critical controls in your application, you can suppress it by un-checking this check box. By default, this check box is not checked.
7. Click **Start Recording**. Silk Test Classic closes the **Record Application State** dialog box and displays one of the following:
 - The **Record Status for the Classic Agent** window, if you are using the Classic Agent.
 - The **Recorder** window, if you are using the Open Agent.The **Status** field flashes *Recording*.
8. Drive your application to the state you want to record. At any point, you can record a verification by pressing **Ctrl+Alt**.
9. When you have finished recording an application state, click **Done** on the **Record Status** window.
Silk Test Classic redisplay the **Record Application State** dialog box. The **Application State Code** field contains the 4Test code that you have recorded. You can take the following actions:

All the information in the window is complete and what you expect.

Click **Paste to Editor** to paste the recorded 4Test code to the editor. Silk Test Classic closes the **Record Application State** dialog box and places the new application state in your file.

You want to alter the code.

Edit the **Application State Code** field.

The application state name is not what you want.

Edit the name in the **Application State Name** field.

The application state on which this application state is based is not the one you want.

Delete the code in the **Application State Code** field, select a new application state from the list, and click **Resume Recording** to re-record the application state.

The application state routine is not finished.

Click **Resume Recording**. Silk Test Classic opens the **Record Status** window.

10. Click **Set Recovery System** to specify the base state for the application under test.

This button is only available when `wMainWindow` has not been set, for example if you have referenced an include file that does not contain `const wMainWindow = MyWin`.

Recording Additional Actions Into an Existing Test

This functionality is supported only if you are using the Open Agent.

Once a test is created, you can open the test and record additional actions to any point in the test. This allows you to update an existing test with additional actions.

1. Open an existing test script.
2. Select the location in the test script into which you want to record additional actions.



Note: Recorded actions are inserted after the selected location. The application under test (AUT) does not return to the base state. Instead, the AUT opens to the scope in which the preceding actions in the test script were recorded.

3. Click **Record > Actions**.

Silk Test Classic minimizes and the **Recording** window opens.

4. Record the additional actions that you want to perform against the AUT.

For information about the actions available during recording, see *Actions Available During Recording*.

5. To stop recording, click **Stop** in the **Recording** window.
6. In the **Record Actions** dialog box, click **Paste to Editor** to insert the recorded actions into your script.
7. Click **Close** to close the **Record Actions** dialog box.

Actions Available During Recording

This functionality is supported only if you are using the Open Agent.

During recording, you can perform the following actions in the **Recording** window:

Action	Steps
Pause recording.	Click Pause to bring the AUT into a specific state without recording the actions, and then click Record to resume recording.
Change the sequence of the recorded actions.	To change the sequence of the recorded actions in the Recording window, select the actions that you want to move and drag them to the new location. To select multiple actions press <code>Ctrl</code> and click on the actions.
Remove a recorded action.	To remove a falsely recorded action from the Recording window, hover the mouse cursor over the action and click Delete this entry .

Action	Steps
Verify an image or a property of a control.	Move the mouse cursor over the object that you want to verify and press Ctrl+Alt . For additional information, see Adding a Verification to a Script while Recording .

Setting Recording Options for the Open Agent


This functionality is supported only if you are using the Open Agent.

You can set the recording options to optimize recording with the Open Agent in the following two ways:

- In the **Recording Options** dialog box.
- Within a script, by using the `SetOption` method.

Using `SetOption` overrides the value specified for the option in the **Recording Options** dialog box. If you do not set an option with `SetOption`, the value specified in the **Recording Options** dialog box is the default.

To set the recording options in the **Recording Options** dialog box, perform the following actions:

1. Click **Options > Recorder**. The **Recording Options** dialog box appears.
2. To set **Ctrl+Shift** as the shortcut key combination to use to pause recording, check the **OPT_ALTERNATE_RECORD_BREAK** check box.
By default, **Ctrl+Alt** is the shortcut key combination.
 **Note:** For SAP applications, you must set **Ctrl+Shift** as the shortcut key combination.
3. To record absolute values for scroll events, check the **OPT_RECORD_SCROLLBAR_ABSOLUT** check box.
4. To record mouse move actions for web applications, Win32 applications, and Windows Forms applications, check the **OPT_RECORD_MOUSEMOVES** check box. You cannot record mouse move actions for child technology domains of the xBrowser technology domain, for example Apache Flex and Swing.
5. If you record mouse move actions, in the **OPT_RECORD_MOUSEMOVE_DELAY** text box, specify how many milliseconds the mouse has to be motionless before a `MouseMove` is recorded.
By default this value is set to 200.
6. To record text clicks instead of clicks, check the **OPT_RECORD_TEXT_CLICK** check box.
Recording text clicks is enabled by default, and is the fallback mechanism when testing applications which display texts. Micro Focus recommends to leave this check box checked.
7. To resize the application under test (AUT) when a recording session starts, check the **OPT_RESIZE_APPLICATION_BEFORE_RECORDING** check box.
This check box is checked by default, enabling the recording sidebar to display next to the AUT. When this check box is unchecked, the AUT and the recording sidebar might overlap.
8. Define custom attributes for recording.
 - a) Select the **Custom Attributes** tab.
 - b) Select the technology domain of the application that you are testing.
For example, to set custom attributes for a web application, select **xBrowser**.
 - c) Add the attributes that you want to use to the list.
Separate attribute names with a comma.

Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index might change when another object is added. If custom attributes are available, the locator generator uses these attributes before any other attribute. The order of the list also represents the priority in which the attributes are used by the locator generator. If the attributes that you specify are not available for the

objects that you select, Silk Test Classic uses the default attributes for the application that you are testing.



Note: You cannot set custom attributes for Apache Flex applications.

9. Set the classes that you want to ignore during recording and replay.

- a) Select the **Transparent Classes** tab.
- b) Add the names of any classes that you want to ignore to the list.

Separate class names with a comma.

10. Specify recording options for web applications.

- a) Select the **Browser** tab.
- b) Add names of attributes that you want to ignore during recording to the **Locator attribute name exclude list**.

For example, if you do not want to record attributes named *height*, add *height* to the list. Separate attribute names with a comma.

- c) Add values of attributes that you want to ignore during recording to the **Locator attribute value exclude list**.

For example, if you do not want to record attributes that have the value *x-auto*, add *x-auto* to the list.

- d) Check the **OPT_XBROWSER_LOWLEVEL** check box to record native user input instead of DOM functions.

For example to record `Click` instead of `DomClick` and `TypeKeys` instead of `SetText`.

If your application uses a plug-in or AJAX, use native user input. If your application does not use a plug-in or AJAX, Micro Focus recommends using high-level DOM functions, which do not require the browser to be focused or active during playback. As a result, tests that use DOM functions are faster and more reliable.

- e) Specify the maximum length for locator attribute values in the **OPT_XBROWSER_LOCATOR_MAX_ATTRIBUTE_VALUE_LENGTH** field.
- f) Check the **OPT_XBROWSER_ENABLE_SMART_CLICK_POSITION** check box to automatically search for a free click spot on the object.

If disabled, the click is always made in the center of the object, and might possibly be performed on an object layered over the target.

11. Specify recording options for WPF applications.

- a) Check the **OPT_WPF_PREFILL_ITEMS** check box to pre-fill items in a `WPFIItemsControl`, for example a `WPFComboBox` or `WPFListBox`, during recording and playback.
- b) Add the names of custom classes that you want to expose during recording and playback to the **Custom WPF class names** list.

12. Click **OK**.

Best Practices for Creating Test Scripts

This functionality is supported only if you are using the Open Agent.

The way in which you write your test cases might have a great impact on the performance and stability of your test set. During recording, Silk Test Classic creates scripts that are as fast and stable as possible. However, there might be circumstances that require you to manually create or edit test scripts. This topic provides some general guidelines that might help you create test scripts that are maintainable, reusable, and lead to stable tests.

- Name your tests consistently and ensure that test names are self-explaining. Try to make the names correspond with the application under test and the tested functionality. For example, the test names *MyApp_SuccessfulLogin* and *MyApp_FailingLogin* are far easier to understand for other users than *Untitled_42* and *Untitled_43*.

- Describe your test cases as thoroughly as possible in a comment. Without a good description of the test case in natural language, someone who needs to change the implementing code might not be able to comprehend what exactly the test is doing.
- Ensure that your application under test is at the proper state when the test case starts. Return the application under test to the correct state before executing the actions in a test case.
- Ensure that your application under test is at a proper state when the test case finishes. If additional tests depend on the outcome of the test, ensure that they can start. Return the application under test to the correct state when the actions in a test case are executed.
- Whenever possible, ensure that your test cases are not depending on the results of other test cases. When this is not possible, ensure that the test cases are executed in the right order.
- Add verifications to your tests, to test the correctness of your application under test as well as the functional flow.
- Use keyword-driven testing to create highly reusable action sets. Bundle commonly used actions into keywords, combine keywords that are often executed sequentially into keyword sequences, and execute combinations of keywords and keyword sequences as keyword driven-tests.
- To keep your tests maintainable and reusable, prefer writing multiple simple test cases that are combinable to writing complex test cases.
- To avoid redundancies in your test set, prefer updating existing test cases to adding new test cases.

Specifying Whether to Use Locators or Tags to Resolve Window Declarations

This functionality is supported only if you are using the Open Agent.



Note: When using the Open Agent to replay tests, you can include locators and tags in the same window declaration. However, tags are deprecated and can no longer be recorded. Micro Focus recommends to use only locators with the Open Agent.

1. Click **Options > General**. The **General Options** dialog box opens.
2. Specify if you want to use locators or tags to resolve window declarations.
 - To use locators to resolve window declarations, check the **Prefer Locator** check box.
 - To use tags to resolve window declarations, uncheck the **Prefer Locator** check box.
3. Click **OK**.

Saving a Script File

To save a script file, click **File > Save**. If it is a new file, Silk Test Classic prompts you for the file name and location.

If you are working within a project, Silk Test Classic prompts you to add the file to the project. Click **Yes** if you want to add the file to the open project, or **No** if you do not want to add this file to the project.

To save a new version of a script's object file when the script file is in view-only mode, choose **File > Save Object File**.

If you are working within a project, you can add the file to your project. If you add object files (.to, .ino) to your project, the files will display under the **Data** node on the **Files** tab. You cannot modify object files within the Silk Test Classic editor because object files are binary. To modify an object file, open the source file (.t or .inc), edit it, and then recompile.

Testing an Application State


Before you run a test case that is associated with an application state, make sure the application state compiles and runs without error.

1. Make the window active that contains the application state and choose **Run > Application State**.
2. On the **Run Application State** dialog box, select the application state you want to run and click **Run**.
If there are compilation errors, Silk Test Classic displays an error window. Fix the errors and rerun the application state.

Editing Remote Locations

This functionality is supported only if you are using the Open Agent.

You can use the **Remote Locations** dialog box to add any browsers and mobile devices on a remote location to the set of applications that you can test.

1. Click **Options > Edit Remote Locations**. The **Remote Locations** dialog box appears.
2. To add an additional remote location, perform the following actions:
 - a) Click on the arrow to the right of **Add Location** to specify whether you want to add a remote location which is using the Silk Test Information Service, or Silk Central.
 **Note:** You can only configure one Silk Central as a remote location. If you have already configured the integration with Silk Central, Silk Central is listed in the remote locations list.
 - b) Click **Add Location**. The **Add Location** dialog box appears.
 - c) Type the URL of the remote location and the port through which Silk Test Classic connects to the information service on the remote machine into the **Host** field.
The default port is 22901.
 - d) *Optional:* Edit the name of the remote location in the **Name** field.
3. To edit an existing remote location, click **Edit**.
4. To remove a remote location, click **Remove**.
5. *Optional:* To reduce the amount of browsers and devices in the **Select Application** dialog, click **Do not show devices and browsers from this location**. The installed browsers and connected devices of the remote location will no longer be displayed in the **Select Application** dialog. By default, all installed browsers and connected devices of all remote locations are displayed in the **Select Application** dialog.
6. By default, remote locations are saved to `%APPDATA%\Silk\SilkTest\conf\remoteLocations.xml`. To change the file in which the remote locations are saved, for example to share the file with other team members, perform the following actions:
 - a) Click **Sharing Options**. The **Sharing Options** dialog box appears.
 - b) Click **Save to shared file**.
 - c) Click **Browse** to select the file.
You could also set the full path to the remote locations in a file named `remoteLocationsFileLocation.properties`, for example if you want to use a file that is located in a repository. For example, to use the remote locations that are specified in the file `remoteLocations.xml`, under `C:\mySources`, create a file with the name `remoteLocationsFileLocation.properties` in the folder `%APPDATA%\Silk\SilkTest\conf` and type `fileLocation=C:\mySources\remoteLocations.xml` into the file.
7. To load a set of already configured remote locations from a shared file, for example to use remote locations that have been specified by another team member, perform the following actions:
 - a) Click **Sharing Options**. The **Sharing Options** dialog box appears.
 - b) Click **Load from shared file**.
 - c) Click **Browse** to select the file.
8. Click **OK**.

When you have added a remote location, the browsers that are installed on the remote location, including Apple Safari on a Mac, are available in the **Web** tab of the **Select Application** dialog box, and the mobile devices that are connected to the remote location are available in the **Mobile** tab of the **Select Application** dialog box.

Locator Spy Dialog Box

This functionality is supported only if you are using the Open Agent.

Use the **Locator Spy** to record a locator keyword or window declarations.

Click **Record > Window Locators**.

Locator	Displays the XPath query string for the object on which the mouse rests when you press Ctrl+Alt or Ctrl+Shift . After capturing the identifier, you can edit it in this box. Silk Test Classic does not verify whether the locator string is unique. We recommend that you ensure that the string is unique. Otherwise additional objects might be found when you run the test. Furthermore, you might want to exclude some of the attributes that Silk Test Classic identifies, because the string will work without them.
Stop Recording Locator	Click to capture the locator that is currently displayed in the Locator field.
Validate Locator	Click to highlight the locator in the test application.
Paste Hierarchy to Editor	Click to paste the window declarations into the open Silk Test Classic file. The hierarchy that displays in the Locator Details section is copied.
Copy Hierarchy to Clipboard	Click to copy the window declarations to the clipboard. Use Edit > Paste to insert the code into a different editing window or to insert it into the current window at the location of your choice. The hierarchy that displays in the Locator Details section is copied.
Paste Locator to Editor	Click to paste the locator into the open Silk Test Classic file. The content of the currently active text field is copied.
Copy Locator to Clipboard	Click to copy the locator to the clipboard. Use Edit > Paste to insert the code into a different editing window or to insert it into the current window at the location of your choice. The content of the currently active text field is copied.

Record Testcase Dialog Box

Use the **Record Testcase** dialog box to record an entire test case, specifying the application state of your choice and including verification statements.

To open the dialog box, click **Record > Testcase** in the Silk Test Classic menu.

Silk Test Classic will compile the currently active script file or test plan, if not already compiled, before opening the **Record Testcase** dialog box. If the current window is not a script file or test plan, Silk Test Classic prompts you to open a script file or create a new script file before opening the dialog box.

Testcase name	Specifies the name of the test case you are going to record. Type the name of the test case or accept the default. The test case name must be less than 128 characters.
Application state	Specifies an application state on which this test case is based. Select an application state from the list. The default is the last application state you specified in this field.
Testcase code	Once you have completed recording the test case, this area displays the 4Test code captured during recording.
Start Recording	Click to close the dialog box. Silk Test Classic displays a message box indicating that the base state is being set and displays one of the following: <ul style="list-style-type: none">• If you are using the Classic Agent, Silk Test Classic displays the Record Status for the Classic Agent window.

- If you are using the Open Agent, Silk Test Classic displays the **Recorder** window.

Paste to Editor	Click to close the dialog box and place the 4Test code, if any, in the associated file.
Set Recovery System	Displays the Set Recovery System dialog box. This button is only available when <code>wMainWindow</code> has not been set. For instance, if you have an include file referenced that does not contain <code>constwMainWindow = MyWin</code> , then the button is available. Once Silk Test Classic sees that <code>wMainWin = Something</code> , the button is disabled.

Update Files Dialog Box

Use to indicate whether you want to update window declarations *on the fly* after you have finished recording.

Click **Paste to Editor** on the **Record** dialog boxes if the recorder discovers new windows or child objects. This dialog box gives you the option of using the generated identifiers or the dynamically instantiated variables.

Paste testcase/appstate only	Select this option to paste the dynamically instantiated variables to the test case or appstate. If you select this option, the Update Window Declaration Detail area is unavailable. If you have clicked the Set Recovery System button on the Record Testcase dialog box or the Record Application State dialog box, this button is disabled. If you have configured your basestate, Silk Test Classic creates a main window declaration and saves it to a <code>.inc</code> file. That means that this option is not available to you.
Paste testcase/appstate and update window declaration(s)	Select this option paste the variables to the test case or appstate AND to add the generated identifiers to a new file or an opened file. If you select this option, the Update Window Declaration Detail area is available.
Update Window Declaration Detail area	This area may contain one or both of the following messages.
Input the file name to store new window objects	Indicate the name of the new file or the name of the referenced file (that is, a file loaded into memory) to contain the window declarations. The file you specify here is listed in the Use Files field on the Runtime Options dialog box.
The window declarations in the following files will be updated	If the recorder finds new child objects, Silk Test Classic lists the files that have changed and updates them after you click OK .

Dynamic Object Recognition

Dynamic object recognition enables you to create test cases that use XPath queries to find and identify objects. Dynamic object recognition uses a `Find` or `FindAll` method to identify an object in a test case. For example, the following query finds the first top-level Shell with the caption SWT Test Application:

```
Desktop.find("/Shell[@caption='SWT Test Application']")
```

To create tests that use dynamic object recognition, you must use the Open Agent.

Examples of the types of test environments where dynamic object recognition works well include:

- In any application environment where the graphical user interface is undergoing changes. For example, to test the Check Me check box in a dialog box that belongs to a menu where the menu and the dialog box name are changing, using dynamic object recognition enables you to test the check box without

concern for what the menu and dialog box are called. You can then verify the check box name, dialog box name, and menu name to ensure that you have tested the correct component.

- In a Web application that includes dynamic tables or text. For example, to test a table that displays only when the user points to a certain item on the web page, use dynamic object recognition to have the test case locate the table without regard for which part of the page needs to be clicked in order for the table to display.
- In an Eclipse environment that uses views. For example, to test an Eclipse environment that includes a view component, use dynamic object recognition to identify the view without regard to the hierarchy of objects that need to open prior to the view.

Using dynamic object recognition compared to using hierarchical object recognition

The benefits of using dynamic object recognition rather than hierarchical object recognition include:

- Dynamic object recognition uses a subset of the XPath query language, which is a common XML-based language defined by the World Wide Web Consortium, W3C. Hierarchical object recognition is based on the concept of a complete description of the application's object hierarchy and as a result is less flexible than dynamic object recognition.
- Dynamic object recognition requires a single object rather than an include file that contains window declarations for the objects in the application that you are testing. Using XPath queries, a test case can locate an object using a `Find` command followed by a supported XPath construct. Hierarchical object recognition uses the include file to identify the objects within the application.

You can create tests for both dynamic and hierarchical object recognition in your test environment. You can use both recognition methods within a single test case if necessary. Use the method best suited to meet your test requirements.

Using dynamic object recognition and window declarations

Silk Test Classic provides an alternative to using `Find` or `FindAll` functions in scripts that use dynamic object recognition. By default, when you record a test case with the Open Agent, Silk Test Classic uses locator keywords in an include (.inc) file to create scripts that use dynamic object recognition and window declarations. Using locator keywords with dynamic object recognition enables users to combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.

Existing test cases that use dynamic object recognition without locator keywords in an INC file will continue to be supported. You can replay these tests, but you cannot record new tests with dynamic object recognition without locator keywords in an INC file. You must manually record test cases that use dynamic object recognition without locator keywords. You can record the XPath query strings to include in test cases by using the **Locator Spy** dialog box.

XPath Basic Concepts

Silk Test Classic supports a subset of the XPath query language. For additional information about XPath, see <http://www.w3.org/TR/xpath20/>.

XPath expressions rely on the current context, the position of the object in the hierarchy on which the `Find` method was invoked. All XPath expressions depend on this position, much like a file system. For example:

- `"//Shell"` finds all shells in any hierarchy starting from the current context.
- `"Shell"` finds all shells that are direct children of the current context.

Additionally, some XPath expressions are context sensitive. For example, `myWindow.find(xpath)` makes `myWindow` the current context.

Silk Test Classic provides an alternative to using `Find` or `FindAll` functions in scripts that use XPath queries. You can use locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Object Type and Search Scope

A locator typically contains the type of object to identify and a search scope. The search scope is one of the following:

- //
- /

Locators rely on the current object, which is the object for which the locator is specified. The current object is located in the object hierarchy of the application's UI. All locators depend on the position of the current object in this hierarchy, much like a file system.

XPath expressions rely on the *current context*, which is the position of the object in the hierarchy on which the `Find` method was invoked. All XPath expressions depend on this position, much like a file system.



Note:

The object type in a locator for an HTML element is either the HTML tag name or the class name that Silk Test Classic uses for this object. For example, the locators `//a` and `//DomLink`, where `DomLink` is the name for hyperlinks in Silk Test Classic, are equivalent. For all non-HTML based technologies only the Silk Test Classic class name can be used.

Example

- `//a` identifies hyperlink objects in any hierarchy relative to the current object.
- `/a` identifies hyperlink objects that are direct children of the current object.



Note: `<a>` is the HTML tag for hyperlinks on a Web page.

Example

The following code sample identifies the first hyperlink in a browser. This example assumes that a variable with the name `browserWindow` exists in the script that refers to a running browser instance. Here the type is "a" and the current object is `browserWindow`.

Using Attributes to Identify an Object

This functionality is supported only if you are using the Open Agent.

To identify an object based on its properties, you can use locator attributes. The locator attributes are specified in square brackets after the type of the object.

Example

The following sample uses the `textContent` attribute to identify a hyperlink with the text `Home`. If there are multiple hyperlinks with the same text, the locator identifies the first one.

```
WINDOW link = WebBrowser.BrowserWindow.Find("//a[@textContent='Home']")
```

Supported XPath Subset

Silk Test Classic supports a subset of the XPath query language. Use a `FindAll` or a `Find` command followed by a supported construct to create a test case.

To create tests that use dynamic object recognition, you must use the Open Agent.

The following table lists the constructs that Silk Test Classic supports.

Supported XPath Construct	Sample	Description
Attribute	<code>MenuItem[@caption='abc']</code>	Finds all menu items with the given caption attribute in their object definition that are children of the current context. The following attributes are supported: <ul style="list-style-type: none"> caption (without caption index) priorlabel (without index) windowid
Index	<code>MenuItem[1]</code>	Finds the first menu item that is a child of the current context. Indices are 1-based in XPath.
Logical Operators: and, or, not, =, !=	<code>MenuItem[not(@caption='a' or @windowid!='b') and @priorlabel='p']</code>	
.	<code>TestApplication.Find("//Dialog[@caption='CheckBox']/../..")</code>	Finds the context on which the Find command was executed. For instance, the sample could have been typed as <code>TestApplication.Find("//Dialog[@caption='CheckBox']")</code> .
..	<code>Desktop.Find("//PushButton[@caption='Previous']/../PushButton[@caption='Ok']")</code>	Finds the parent of an object. For instance, the sample finds a PushButton with the caption "Ok" that has a sibling PushButton with the caption "Previous."
/	<code>/Shell</code>	Finds all shells that are direct children of the current object. "/Shell" is equivalent to "/Shell" and "Shell".
/	<code>/Shell/MenuItem</code>	Finds all menu items that are a child of the current object.
//	<code>//Shell</code>	Finds all shells in any hierarchy relative to the current object.
//	<code>//Shell/MenuItem</code>	Finds all menu items that are direct or indirect children of a Shell that is a direct child of the current object.
//	<code>//MenuItem</code>	Finds all menu items that are direct or indirect children of the current context.
*	<code>*[@caption='c']</code>	Finds all objects with the given caption that are a direct child of the current context.
*	<code>./MenuItem/*/Shell</code>	Finds all shells that are a grandchild of a menu item.

The following table lists the XPath constructs that Silk Test Classic does not support.

Unsupported XPath Construct	Example
Comparing two attributes with each other.	<code>PushButton[@caption = @windowid]</code>
An attribute name on the right side is not supported. An attribute name must be on the left side.	<code>PushButton['abc' = @caption]</code>
Combining multiple XPath expressions with 'and' or 'or'.	<code>PushButton [@caption = 'abc'] or .// Checkbox</code>
More than one set of attribute brackets.	<code>PushButton[@caption = 'abc'] [@windowid = '123']</code> Use <code>PushButton [@caption = 'abc and @windowid = '123']</code> instead.
More than one set of index brackets.	<code>PushButton[1][2]</code>
Any construct that does not explicitly specify a class or the class wildcard, such as including a wildcard as part of a class name.	<code>//[@caption = 'abc']</code> Use <code>//*[@caption = 'abc']</code> instead. <code>//*Button[@caption='abc']"</code>

XPath Samples

The following table lists sample XPath queries and explains the semantics for each query.

XPath String	Description
<code>desktop.Find("/Shell[@caption='SWT Test Application'] ")</code>	Finds the first top-level <code>Shell</code> with the given caption.
<code>desktop.Find("// MenuItem[@caption='Control']")</code>	Finds the <code>MenuItem</code> in any hierarchy with the given caption.
<code>myShell.Find("//MenuItem[@caption! ='Control']")</code>	Finds an <code>MenuItem</code> in any child hierarchy of <i>myShell</i> that does not have the given caption.
<code>myShell.Find("Menu[@caption='Control'] /MenuItem[@caption!='Control']")</code>	Looks for a specified <code>MenuItem</code> with the specified <code>Menu</code> as parent that has <i>myShell</i> as parent.
<code>myShell.Find("// MenuItem[@caption='Control' and @windowid='20']")</code>	Finds a <code>MenuItem</code> in any child hierarchy of <i>myWindow</i> with the given caption and <code>windowId</code> .
<code>myShell.Find("// MenuItem[@caption='Control' or @windowid='20']")</code>	Finds a <code>MenuItem</code> in any child hierarchy of <i>myWindow</i> with the given caption or <code>windowId</code> .
<code>desktop.FindAll("/Shell[2]/** PushButton")</code>	Finds all <code>PushButtons</code> that have an arbitrary parent that has the second top-level shell as parent.
<code>desktop.FindAll("/Shell[2]// PushButton")</code>	Finds all <code>PushButtons</code> that use the second shell as direct or indirect parent.
<code>myBrowser.Find("//FlexApplication[1]// FlexButton[@caption='ok']")</code>	Looks up the first <code>FlexButton</code> within the first <code>FlexApplication</code> within the given browser.

XPath String	Description
<code>myBrowser.FindAll("//td[@class='abc*']//a[@class='xyz']")</code>	Finds all link elements with attribute class <code>xyz</code> that are direct or indirect children of <code>td</code> elements with attribute class <code>abc*</code> .

Supported Locator Attributes

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests. If necessary, you can change the attribute type in one of the following ways:

- Manually typing another attribute type and value.
- Specifying another preference for the default attribute type by changing the custom attributes list values.

To create tests that use locators, you must use the Open Agent.

Using Locators

Within Silk Test Classic, literal references to identified objects are referred to as *locators*. For convenience, you can use shortened forms for the locator strings in scripts. Silk Test Classic automatically expands the syntax to use full locator strings when you playback a script. When you manually code a script, you can omit the following parts in the following order:

- The search scope, `//`.
- The object type name. Silk Test Classic defaults to the class name.
- The surrounding square brackets of the attributes, `[]`.

When you manually code a script, we recommend that you use the shortest form available.



Note: When you identify an object, the full locator string is captured by default.

The following locators are equivalent:

- The first example uses the full locator string.
To confirm the full locator string, use the dialog box.
- The second example works when the browser window already exists. Alternatively, you can use the shortened form.

To find an object that has no real attributes for identification, use the index. For instance, to select the second hyperlink on a Web page, you can type:

Additionally, to find the first object of its kind, which might be useful if the object has no real attributes, you can type:

Using Locators to Check if an Object Exists

You can use the `Exists` method to determine if an object exists in the application under test.

The following code checks if a hyperlink with the text *Log out* exists on a Web page:

```
if (browserWindow.Exists( "//a[@textContents='Log out']" )) {
    // do something
}
```

Using the Find method

You can use the `Find` method and the `FindOptions` method to check if an object, which you want to use later, exists.

The following code searches for a window and closes the window if the window is found:

```
Window mainWindow = desktop.Find("//Window[@caption='My Window']", new
FindOptions(false))
if (mainWindow != null){
    mainWindow.CloseSynchron()
}
```

Identifying Multiple Objects with One Locator

You can use the `FindAll` method to identify all objects that match a locator rather than only identifying the first object that matches the locator.

Example

The following code example uses the `FindAll` method to retrieve all hyperlinks of a Web page:

```
LIST OF DOMLINK links = browserWindow.FindAll("//a")
```

Locator Customization

This section describes how you can create stable locators that enable Silk Test Classic to reliably recognize the controls in your application under test (AUT).

Silk Test Classic relies on the identifiers that the AUT exposes for its UI controls and is very flexible and powerful in regards to identifying UI controls. Silk Test Classic can use any declared properties for any UI control class and can also create locators by using the hierarchy of UI controls. From the hierarchy, Silk Test Classic chooses the most appropriate items and properties to identify each UI control.

Silk Test Classic can exclude dynamic numbers of controls along the UI control hierarchy, which makes the object recognition in Silk Test Classic very robust against changes in the AUT. Intermediate grouping controls that change the hierarchy of the UI control tree, like formatting elements in Web pages, can be excluded from the object recognition.

Some UI controls do not expose meaningful properties, based on which they can be identified uniquely. Applications which include such controls are described as applications with *bad testability*. Hierarchies, and especially dynamic hierarchies, provide a good means to create unique locators for such applications. Applications with *good testability* should always provide a simple mechanism to identify UI controls uniquely.

One of the simplest and most effective practices to make your AUT easier to test is to introduce stable identifiers for controls and to expose these stable identifiers through the existing interfaces of the application.

Stable Identifiers

A *stable identifier* for a UI control is an identifier that does not change between invocations of the control and between different versions of the application, in which the UI control exists. A stable identifier needs to be unique in the context of its usage, meaning that no other control with the same identifier is accessible at the same time. This does not necessarily mean that you need to use GUID-style identifiers that are unique in a global context. Identifiers for controls should be readable and provide meaningful names. Naming conventions for these identifiers will make it much easier to associate the identifier to the actual control.

Example: Is the caption a good identifier for a control?

Very often test tools are using the *caption* as the default identifier for UI controls. The caption is the text in the UI that is associated with the control. However, using the caption to identify a UI control has the following drawbacks:

- The caption is not stable. Captions can change frequently during the development process. For example, the UI of the AUT might be reviewed at the end of the development process. This prevents introducing UI testing early in the development process because the UI is not stable.
- The caption is not unique. For example, an application might include multiple buttons with the caption **OK**.
- Many controls are not exposing a caption, so you need to use another property for identification.
- Using captions for testing localized applications is cumbersome, as you need to maintain a caption for a control in each language and you also have to maintain a complex script logic where you dynamically can assign the appropriate caption for each language.

Creating Stable Locators

One of the main advantages of Silk Test Classic is the flexible and powerful object-recognition mechanism. By using XPath notation to locate UI controls, Silk Test Classic can reliably identify UI controls that do not have any suitable attributes, as long as there are UI elements near the element of interest that have suitable attributes. The XPath locators in Silk Test Classic can use the entire UI control hierarchy or parts of it for identifying UI controls. Especially modern AJAX toolkits, which dynamically generate very complex Document Object Models (DOMs), do not provide suitable control attributes that can be used for locating UI controls.

In such a case, test tools that do not provide intelligent object-recognition mechanisms often need to use index-based recognition techniques to identify UI controls. For example, identify the *n*-th control with icon *Expand*. This often results in test scripts that are hard to maintain, as even minor changes in the application can break the test script.

A good strategy to create stable locators for UI controls that do not provide useful attributes is to look for an anchor element with a stable locator somewhere in the hierarchy. From that anchor element you can then work your way to the element for which you want to create the locator.

Silk Test Classic uses this strategy when creating locators, however there might be situations in which you have to manually create a stable locator for a control.

Example: Locating Siblings of a Control

This functionality is supported only if you are using the Open Agent.

This topic describes how you can locate a control, which does not provide any meaningful attributes that can be used in locators, when a stable locator for a sibling of the control is available.

Assume that you have already identified the control **Item 0.0**, which has the following stable locator:

```
/BrowserApplication//BrowserWindow//DIV[@textContent='Item 0.0']
```

If you know that **Item 0.0** has a sibling of the type *a*, you can use the following code to build a stable locator for the sibling:

```
/BrowserApplication//BrowserWindow//DIV[@textContent='Item 0.0']/../a
```

You can also use the sibling approach to identify text fields. Text fields often do not provide any meaningful attributes that can be used in locators. By using the label of a text field, you could create a meaningful locator for the text field, because the label is the best identifier for the text field from the perspective of a tester. You can easily use the label as a part of the locator for a test field by using the sibling approach. For example, if the text field is a sibling of a label with the text **User Name**, you can use the following locator:

```
/BrowserApplication//BrowserWindow//DIV[@textContent='User Name']/../input[@type='text']
```

Example: Locating the Expand Icon in a Dynamic GWT Tree

This functionality is supported only if you are using the Open Agent.

The Google Widget Toolkit (GWT) is a very popular and powerful toolkit, which is hard to test. The dynamic tree control is a very commonly used UI control in GWT. To expand the tree, we need to identify the **Expand** icon element.

You can find a sample dynamic GWT tree at <http://samples.gwtproject.org/samples/Showcase/Showcase.htm##!CwTree>.

The default locator generated by Silk Test Classic is the following:

```
/BrowserApplication//BrowserWindow//DIV[@id='gwt-debug-cwTree-dynamicTree-root-child0']/DIV/DIV[1]//IMG[@border='0']
```

For the following reasons, this default locator is no reliable locator for identifying the **Expand** icon for the control **Item 0.0**:

- The locator is complex and built on multiple hierarchies. A small change in the DOM structure, which is dynamic with AJAX, can break the locator.
- The locator contains an index for some of the controls along the hierarchy. Index based locators are generally weak as they find controls by their occurrence, for example finding the sixth expand icon in a tree does not define the control well. An exception to that rule would be if the index is used to express different data sets that you want to identify, for example the sixth data row in a grid.

Often a good strategy for finding better locators is to search for siblings of elements that you need to locate. If you find siblings with better locators, XPath allows you to construct the locator by identifying those siblings. In this case, the tree item **Item 0.0** provides a better locator than the **Expand** icon. The locator of the tree item **Item 0.0** is a stable and simple locator as it uses the `@textContent` property of the control.

By default, Silk Test Classic uses the property `@id`, but in GWT the `@id` is often not a stable property, because it contains a value like `'gwt-uid-<nnn>'`, where `<nnn>` changes frequently, even for the same element between different calls.

You can manually change the locator to use the `@textContent` property instead of the `@id`.

Original Locator:

```
/BrowserApplication//BrowserWindow//DIV[@id='gwt-uid-109']
```

Alternate Locator:

```
/BrowserApplication//BrowserWindow//DIV[@textContent='Item 0.0']
```

Or you can instruct Silk Test Classic to avoid using `@id='gwt-uid-<nnn>'`. In this case Silk Test Classic will automatically record the stable locator. You can do this by adding the text pattern that is used in `@id` properties to the locator attribute value blacklist. In this case, add `gwt-uid*` to the blacklist.

When inspecting the hierarchy of elements, you can see that the control **Item 0.0** and the **Expand** icon control have a joint root node, which is a `DomTableRow` control.

To build a stable locator for the **Expand** icon, you first need to locate **Item 0.0** with the following locator:

```
/BrowserApplication//BrowserWindow//DIV[@textContent='Item 0.0']
```

Then you need to go up two levels in the element hierarchy to the `DomTableRow` element. You express this with XPath by adding `../../` to the locator. Finally you need to search from `DomTableRow` for the **Expand** icon. This is easy as the **Expand** icon is the only `IMG` control in the sub-tree. You express this with XPath by adding `//IMG` to the locator. The final stable locator for the **Expand** icon looks like the following:

```
/BrowserApplication//BrowserWindow//DIV[@textContent='Item 0.0']/../..///IMG
```

Custom Attributes

This functionality is supported only if you are using the Open Agent.

Add custom attributes to a test application to make a test more stable. You can use custom attributes with the following technologies:

- Java SWT
- Swing
- WPF
- xBrowser
- Windows Forms
- SAP

For example, in Java SWT, the developer implementing the GUI can define an attribute (for example, `silkTestAutomationId`) for a widget that uniquely identifies the widget in the application. A tester using Silk Test Classic can then add that attribute to the list of custom attributes (in this case, `silkTestAutomationId`), and can identify controls by that unique ID. Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index will change whenever another widget is added before the one you have defined already.

If more than one object is assigned the same custom attribute value, all the objects with that value will return when you call the custom attribute. For example, if you assign the unique ID, 'loginName' to two different fields, both fields will return when you call the `loginName` attribute.

First, enable custom attributes for your application and then create the test.

Recording tests that use dynamic object recognition

Using custom class attributes becomes even more powerful when it is used in combination with dynamic object recognition. For example, if you create a button in the application that you want to test using the following code:

```
Button myButton = Button(parent, SWT.NONE);
myButton.setData("SilkTestAutomationId", "myButtonId");
```

To add the attribute to your XPath query string in your test case, you can use the following query:

```
Window button = Desktop.Find("../
PushButton[@SilkTestAutomationId='myButton']")
```

Custom Attributes for Apache Flex Applications

Apache Flex applications use the predefined property `automationName` to specify a stable identifier for the Apache Flex control as follows:

```
<?xml version="1.0" encoding="utf-8"?>
  <s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" width="400" height="300">
    <fx:Script>
      ...
    </fx:Script>
    <s:Button x="247" y="81" label="Button" id="button1" enabled="true"
click="button1_clickHandler(event)"
      automationName="AID_buttonRepeat"/>
    <s:Label x="128" y="123" width="315" height="18" id="label1"
verticalAlign="middle"
      text="awaiting your click" textAlign="center"/>
  </s:Group>
```

Apache Flex application locators look like the following:

```
...//SparkApplication//SparkButton[@caption='AID_buttonRepeat']
```



Attention: For Apache Flex applications, the *automationName* is always mapped to the locator attribute *caption* in Silk Test Classic. If the *automationName* attribute is not specified, Silk Test Classic maps the property *ID* to the locator attribute *caption*.

Java SWT Custom Attributes

You can add custom attributes to a test application to make a test more stable. For example, in Java SWT, the developer implementing the GUI can define an attribute (for example, 'silkTestAutomationId') for a widget that uniquely identifies the widget in the application. A tester using Silk Test Classic can then add that attribute to the list of custom attributes (in this case, 'silkTestAutomationId'), and can identify controls by that unique ID. Using a custom attribute is more reliable than other attributes like *caption* or *index*, since a *caption* will change when you translate the application into another language, and the *index* will change whenever another widget is added before the one you have defined already.

If more than one object is assigned the same custom attribute value, all the objects with that value will return when you call the custom attribute. For example, if you assign the unique ID, 'loginName' to two different text fields, both fields will return when you call the 'loginName' attribute.

Java SWT Example

If you create a button in the application that you want to test using the following code:

```
Button myButton = Button(parent, SWT.NONE);  
myButton.setData("SilkTestAutomationId", "myButtonId");
```

To add the attribute to your XPath query string in your test, you can use the following query:

```
Dim button =  
desktop.PushButton("@SilkTestAutomationId='myButton' ")
```

To enable a Java SWT application for testing custom attributes, the developers must include custom attributes in the application. Include the attributes using the `org.swt.widgets.Widget.setData(String key, Object value)` method.

Custom Attributes for Web Applications

HTML defines a common attribute *ID* that can represent a stable identifier. By definition, the *ID* uniquely identifies an element within a document. Only one element with a specific *ID* can exist in a document.

However, in many cases, and especially with AJAX applications, the *ID* is used to dynamically identify the associated server handler for the HTML element, meaning that the *ID* changes with each creation of the Web document. In such a case the *ID* is not a stable identifier and is not suitable to identify UI controls in a Web application.

A better alternative for Web applications is to introduce a new custom HTML attribute that is exclusively used to expose UI control information to Silk Test Classic.

Custom HTML attributes are ignored by browsers and by that do not change the behavior of the AUT. They are accessible through the DOM of the browser. Silk Test Classic allows you to configure the attribute that you want to use as the default attribute for identification, even if the attribute is a custom attribute of the control class. To set the custom attribute as the default identification attribute for a specific technology domain, click **Options > Recorder > Custom Attributes** and select the technology domain.

The application developer just needs to add the additional HTML attribute to the Web element.

Original HTML code:

```
<A HREF="http://abc.com/control=4543772788784322..." <IMG  
src="http://abc.com/xxx.gif" width=16 height=16> </A>
```

HTML code with the new custom HTML attribute *AUTOMATION_ID*:

```
<A HREF="http://abc.com/control=4543772788784322..."  
AUTOMATION_ID = "AID_Login" <IMG src="http://abc.com/xxx.gif"  
width=16 height=16> </A>
```

When configuring the custom attributes, Silk Test Classic uses the custom attribute to construct a unique locator whenever possible. Web locators look like the following:

```
...//DomLink[@AUTOMATION_ID='AID_Login']
```

Example: Changing ID

One example of a changing ID is the Google Widget Toolkit (GWT), where the ID often holds a dynamic value which changes with every creation of the Web document:

```
ID = 'gwt-uid-<nnn>'
```

In this case `<nnn>` changes frequently.

Custom Attributes for Windows Forms Applications

Windows Forms applications use the predefined automation property `automationId` to specify a stable identifier for the Windows forms control.

Silk Test Classic automatically will use this property for identification in the locator. Windows Forms application locators look like the following:

```
/FormsWindow//PushButton[@automationId='btnBasicControls']
```

Custom Attributes for WPF Applications

WPF applications use the predefined automation property `AutomationProperties.AutomationId` to specify a stable identifier for the WPF control as follows:

```
<Window x:Class="Test.MainWindow"  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
Title="MainWindow" Height="350" Width="525">  
<Grid>  
  <Button AutomationProperties.AutomationId="AID_buttonA">The  
Button</Button>  
</Grid>  
</Window>
```

Silk Test Classic automatically uses this property for identification in the locator. WPF application locators look like the following:

```
/WPFWindow[@caption='MainWindow']//WPFButton[@automationId='AID_buttonA']
```

Troubleshooting Performance Issues for XPath

When testing applications with a complex object structure, for example complex web applications, you may encounter performance issues, or issues related to the reliability of your scripts. This topic describes how you can improve the performance of your scripts by using different locators than the ones that Silk Test Classic has automatically generated during recording.



Note: In general, we do not recommend using complex locators. Using complex locators might lead to a loss of reliability for your tests. Small changes in the structure of the tested application can break such a complex locator. Nevertheless, when the performance of your scripts is not satisfying, using more specific locators might result in tests with better performance.

The following is a sample element tree for the application MyApplication:

```
Root  
  Node id=1
```



```
Leaf id=2
Leaf id=3
Leaf id=4
Leaf id=5
Node id=6
Node id=7
  Leaf id=8
  Leaf id=9
Node id=9
  Leaf id=10
```

You can use one or more of the following optimizations to improve the performance of your scripts:

- If you want to locate an element in a complex object structure, search for the element in a specific part of the object structure, not in the entire object structure. For example, to find the element with the identifier 4 in the sample tree, if you have a query like `Root.Find("//Leaf[@id='4']")`, replace it with a query like `Root.Find("/Node[@id='1']/Leaf[@id='4']")`. The first query searches the entire element tree of the application for leaves with the identifier 4. The first leaf found is then returned. The second query searches only the first level nodes, which are the node with the identifier 1 and the node with the identifier 6, for the node with the identifier 1, and then searches in the subtree of the node with the identifier 1 for all leaves with the identifier 4.
- When you want to locate multiple items in the same hierarchy, first locate the hierarchy, and then locate the items in a loop. If you have a query like `Root.FindAll("/Node[@id='1']/Leaf")`, replace it with a loop like the following:

```
testcase Test() appstate none
  WINDOW node
  INTEGER i

  node = Root.Find("/Node[@id='1']")
  for i = 1 to 4 step 1
    node.Find("/Leaf[@id='{i}']")
```

Highlighting Objects During Recording

During recording, the active object in the AUT is highlighted by a green rectangle. As soon as a new object becomes active this new object is highlighted. If the same object remains active for more than 0.5 seconds a tool-tip will be displayed that displays the class name of the active object and also the current position of the mouse relative to the active object. This tool-tip will no longer be displayed when a new object becomes active, the user presses the mouse, or automatically after 2 seconds.

Overview of the Locator Keyword

Traditional Silk Test Classic scripts that use the Classic Agent use hierarchical object recognition. When you record a script that uses hierarchical object recognition, Silk Test Classic creates an include (.inc) file that contains window declarations and tags for the GUI objects that you are testing. Essentially, the INC file serves as a central global, repository of information about the application under test. It contains all the data structures that support your test cases and test scripts.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. The locator is the actual name of the object, as opposed to the identifier, which is the logical name. Silk Test Classic uses the locator to identify objects in the application when executing test cases. Test cases never use the locator to refer to an object; they always use the identifier.

You can also manually create test cases that use dynamic object recognition without locator keywords. Dynamic object recognition uses a `Find` or `FindAll` function and an XPath query to locate the objects that you want to test. No include file, window declaration, or tags are required.

The advantages of using locators with an INC file include:

- You combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.
- Enhancing legacy INC files with locators facilitates a smooth transition from using hierarchical object recognition to new scripts that use dynamic object recognition. You use dynamic object recognition but your scripts look and feel like traditional, Silk Test Classic tag-based scripts that use hierarchical object recognition.
- You can use `AutoComplete` to assist in script creation. `AutoComplete` requires an INC file.

Syntax

The syntax for the locator keyword is:

```
[gui-specifier] locator locator-string
```

where `locator-string` is an XPath string. The XPath string is the same locator string that is used for the `Find` or `FindAll` functions.

Example

The following example shows a window declaration that uses locators:

```
[ - ] window MainWin TestApplication
  [ ] locator "//MainWin[@caption='Test Application']"
  [ ]
  [ ] // The working directory of the application when it is
invoked
  [ ] const sDir = "{SYS_GetEnv("SEGUE_HOME")}"
  [ ]
  [ ] // The command line used to invoke the application
  [ ] const sCmdLine =
"" "{SYS_GetEnv("SEGUE_HOME")}testapp.exe""
  [ ]
  [ - ] MenuItem Control
    [ ] locator "//MenuItem[@caption='Control']"
  [ - ] MenuItem CheckBox
    [ ] locator "//MenuItem[@caption='Check box']"
  [ - ] MenuItem ComboBox
    [ ] locator "//MenuItem[@caption='Combo box']"
  [ - ] MenuItem ListBox
    [ ] locator "//MenuItem[@caption='List box']"
  [ - ] MenuItem PopupList
    [ ] locator "//MenuItem[@caption='Popup list']"
  [ - ] MenuItem PushButton
    [ ] locator "//MenuItem[@caption='Push button']"
  [ - ] MenuItem RadioButton
    [ ] locator "//MenuItem[@caption='Radio button']"
  [ - ] MenuItem ListView
    [ ] locator "//MenuItem[@caption='List view']"
  [ - ] MenuItem PageList
    [ ] locator "//MenuItem[@caption='Page list']"
  [ - ] MenuItem UpDown
    [ ] locator "//MenuItem[@caption='Up-Down']"
  [ - ] MenuItem TreeView
    [ ] locator "//MenuItem[@caption='Tree view']"
  [ - ] MenuItem Textfield
    [ ] locator "//MenuItem[@caption='Textfield']"
  [ - ] MenuItem StaticText
    [ ] locator "//MenuItem[@caption='Static text']"
  [ - ] MenuItem TrackBar
    [ ] locator "//MenuItem[@caption='Track bar']"
  [ - ] MenuItem ToolBar
```

```

[ ] locator "//MenuItem[@caption='Tool bar']"
[-] MenuItem Scrollbar
[ ] locator "//MenuItem[@caption='Scrollbar']"
[ ]
[-] DialogBox CheckBox
[ ] locator "//DialogBox[@caption='Check Box']"
[-] CheckBox TheCheckBox
[ ] locator "//CheckBox[@caption='The check box']"
[-] PushButton Exit
[ ] locator "//PushButton[@caption='Exit']"

```

For example, if the script uses a menu item like this:

```
TestApplication.Control.TreeView.Pick()
```

Then the menu item is resolved by using dynamic object recognition Find calls using XPath locator strings.

The above statement is equivalent to:

```
Desktop.Find("//MainWin[@caption='Test Application']
//Menu[@caption='Control']//MenuItem[@caption='Tree
view']").Pick()
```

Locator String Syntax

For convenience, you can use shortened forms for the XPath locator strings. Silk Test Classic automatically expands the syntax to use full XPath strings when you run a script. You can omit:

- The hierarchy separator, “//”. Silk Test Classic defaults to using “//”.
- The class name. Silk Test Classic defaults to the class name of the window that contains the locator.
- The surrounding square brackets of the attributes, “[]”.
- The “@caption=” if the XPath string refers to the caption.

The following locators are equivalent:

```
Menu Control
//locator "//Menu[@caption='Control']"
//locator "Menu[@caption='Control']"
//locator "[@caption='Control']"
//locator "@caption='Control' "
locator "Control"
```

You can use shortened forms for the XPath locator strings only when you use an INC file. For scripts that use dynamic object recognition without an INC file, you must use full XPath strings.

Window Hierarchies

You can create window hierarchies without locator strings. In the following example, the “Menu Control” acts only as a logical hierarchy, used to provide the INC file with more structure. “Menu Control” does not contribute to finding the elements further down the hierarchy.

```
[-] window MainWin TestApplication
[ ] locator "//MainWin[@caption='Test Application']"
[-] Menu Control
[-] MenuItem TreeView
[ ] locator "//MenuItem[@caption='Tree view']"
```

In this case, the statement:

```
TestApplication.Control.TreeView.Pick()
```

is equivalent to:

```
Desktop.Find("//MainWin[@caption='Test Application']
//MenuItem[@caption='Tree view']").Pick()
```

Window Declarations

A window declaration in Silk Test Classic cannot be executed for both agent types, Classic Agent and Open Agent, during the execution of a test. The window declaration will only be executed for one of the agent types.

Expressions

You can use expressions in locators. For example, you can specify:

```
[ - ] STRING getSWTVersion()  
    [ ] return SYS_GETENV("SWT_VERSION")  
[ - ] window Shell SwtTestApplication  
    [ ] locator "SWT {getSWTVersion()} Test Application"
```

Comparing the Locator Keyword to the Tag Keyword

The syntax of locators is identical to the syntax of the tag keyword.

The overall rules for locators are the same as for tags. There can be only one locator per window, except for different gui-specifiers, in this case there can be only one locator per gui-specifier.

You can use expressions in locators and tags.

The locator keyword requires a script that uses the Open Agent while the tag keyword requires a script that uses the Classic Agent.

Test Cases

This section describes how you can use automated tests to address single objectives of a test plan.

Overview of Test Cases

A test case is an automated test that addresses one objective of a test plan. A test case:

- Drives the application from the initial state to the state you want to test.
- Verifies that the actual state matches the expected (correct) state. Your QA department might use the term baseline to refer to this expected state. This stage is the heart of the test case.
- Cleans up the application, in preparation for the next test case, by undoing the steps performed in the first stage.

In order for a test case to function properly, the application must be in a stable state when the test case begins to execute. This stable state is called the base state. The recovery system is responsible for maintaining the base state in the event the application fails or crashes, either during the execution of a test cases or between test cases.

Each test case is independent and should perform its own setup, driving the application to the state that you want to test, executing the test case, and then returning the application to the base state. The test case should not rely on the successful or unsuccessful completion of another test case, and the order in which the test case is executed should have no bearing on its outcome. If a test case relies on a prior test case to perform some setup actions, and an error causes the setup to fail or, worse yet, the application to crash, all subsequent test cases will fail because they cannot achieve the state where the test is designed to begin.

A test case has a single purpose: a single test case should verify a single aspect of the application. When a test case designed in this manner passes or fails, it is easy to determine specifically what aspect of the target application is either working or not working.

If a test case contains more than one objective, many outcomes are possible. Therefore, an exception may not point specifically to a single failure in the software under test but rather to several related function points. This makes debugging more difficult and time-consuming and leads to confusion in interpreting and

quantifying results. The result is an overall lack of confidence in any statistics that might be generated. But there are techniques you can use to perform more than one verification in a test case.

Types of test cases

Silk Test Classic supports two types of test cases, depending on the type of application that you are testing. You can create test cases that use:

Hierarchical object recognition	This is a fast, easy method for creating scripts. This type of testing is supported for all application types.
Dynamic object recognition	This is a more robust and easy to maintain method for creating scripts. However, dynamic object recognition is only supported for applications that use the Open Agent.

If you are using the Open Agent, you can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements. You can use both recognition methods within a single test case if necessary.

Anatomy of a Basic Test Case

A test case is comprised of testcase keywords and object-oriented commands. You place a group of test cases for an application into a file called a script.

Each automated test for an application begins with the testcase keyword, followed by the name of the test case. The test case name should indicate the type of testing being performed.

The core of the test case is object-oriented 4Test commands that drive, verify, and clean up your application. For example, consider this command:

```
TextEditor.File.New.Pick
```

The first part of the command, `TextEditor.File.New`, is the name of a GUI object. The last part of the command, `Pick`, is the operation to perform on the GUI object. The dot operator (.) delimits each piece of the command. When this command is executed at runtime, it picks the **New** menu item from the **File** menu of the Text Editor application.

Types of Test Cases

There are two basic types of test cases:

- Level 1 tests, often called smoke tests or object tests, verify that an application's GUI objects function properly. For example, they verify that text boxes can accept keystrokes and check boxes can display a check mark.
- Level 2 tests verify an application feature. For example, they verify that an application's searching capability can correctly find different types of search patterns.

You typically run Level 1 tests when you receive a new build of your application, and do not run Level 2 tests until your Level 1 tests achieve a specific pass/fail ratio. The reason for this is that unless your application's graphical user interface works, you cannot actually test the application itself.

Test Case Design

When defining test requirements, the goal is to vigorously test each application feature. To do so, you need to decide which set of inputs to a feature will provide the most meaningful test results.

As you design your test cases, you may want to associate data with individual objects, which can then be referenced inside test cases. You may find this preferable to declaring global variables or passing parameters to your test cases.

The type of data you decide to define within a window declaration will vary, depending on the type of testing you are doing. Some examples include:

- The default value that you expect the object to have when it displays.
- The tab sequence for each of a dialog box's child objects.

The following declaration for the **Find** dialog contains a list that specifies the tab sequence of the dialog box children.

```
window DialogBox Find
  tag "Find"
  parent TextEditor
  LIST OF WINDOW lwTabOrder = {...}
    FindWhat
    CaseSensitive
    Direction
    Cancel
```

For more information about the syntax to use for lists, see *LIST data type*.

Before you begin to design and record test cases, make sure that the built-in recovery system can close representative dialogs from your application window.

Constructing a Test Case

This section explains the methodology you use when you design and record a test case.

A test case has three stages

Each test case that you record should have the following stages:

- Stage 1** The test case drives the application from the initial state to the state you want to test.
- Stage 2** The test case verifies that the actual state matches the expected (correct) state. Your QA department might use the term baseline to refer to this expected state. This stage is the heart of the test case.
- Stage 3** The test case cleans up the application, in preparation for the next test case, by undoing the steps performed in stage 1.

Each test case is independent

Each test case you record should perform its own setup in stage 1, and should undo this setup in stage 3, so that the test case can be executed independently of every other test case. In other words, the test case should not rely on the successful or unsuccessful completion of another test case, and the order in which it is executed should have no bearing on its outcome.

If a test case relies on a prior test case to perform some setup actions, and an error causes the setup to fail or, worse yet, the application to crash, all subsequent test cases will fail because they cannot achieve the state where the test is designed to begin.

A test case has a single purpose

Each test case you record should verify a single aspect of the application in stage 2. When a test case designed in this manner passes or fails, it's easy to determine specifically what aspect of the target application is either working or not working.

If a test case contains more than one objective, many outcomes are possible. Therefore, an exception may not point specifically to a single failure in the software under test but rather to several related function points. This makes debugging more difficult and time-consuming and leads to confusion in interpreting and quantifying results. The net result is an overall lack of confidence in any statistics that might be generated.

There are techniques you can use to do more than one verification in a test case.

A test case starts from a base state

In order for a test case to be able to function properly, the application must be in a stable state when the test case begins to execute. This stable state is called the base state. The recovery system is responsible for maintaining the base state in the event the application fails or crashes, either during a test case's execution or between test cases.

DefaultBaseState

To restore the application to the base state, the recovery system contains a routine called `DefaultBaseState` that makes sure that:

- The application is running and is not minimized.
- All other windows, for example dialog boxes, are closed.
- The main window of the application is active.

If these conditions are not sufficient for your application, you can customize the recovery system.

Defining test requirements

When defining test requirements, the goal is to rigorously test each application feature. To do so, you need to decide which set of inputs to a feature will provide the most meaningful test results.

Data in Test Cases

What data does the feature expect

A user can enter three pieces of information in the **Find** dialog box:

- The search can be case sensitive or insensitive, depending on whether the **Case Sensitive** check box is checked or unchecked.
- The search can be forward or backward, depending on whether the **Down** or **Up** option button is selected.
- The search can be for any combination of characters, depending on the value entered in the **Find What** text box.

Create meaningful data combinations

To organize this information, it is helpful to construct a table that lists the possible combinations of inputs. From this list, you can then decide which combinations are meaningful and should be tested. A partial table for the **Find** dialog box is shown below:

Case Sensitive	Direction	Search String
Yes	Down	Character
Yes	Down	Partial word (start)
Yes	Down	Partial word (end)
Yes	Down	Word
Yes	Down	Group of words
Yes	Up	Character
Yes	Up	Partial word (start)
Yes	Up	Partial word (end)
Yes	Up	Word

Case Sensitive	Direction	Search String
Yes	Up	Group of words

Saving Test Cases

When saving a test case, Silk Test Classic does the following:

- Saves a source file, giving it the `.t` extension; the source file is an ASCII text file, which you can edit.
- Saves an object file, giving it the `.to` extension; the object file is a binary file that is executable, but not readable by you.

For example, if you name a test case (script file) `mytests` and save it, you will end up with two files: the source file `mytests.t`, in the location you specify, and the object file `mytests.to`.

To save a new version of a script's object file when the script file is in view-only mode, click **File > Save Object File**.

Recording Without Window Declarations

If you record a test case against a GUI object for which there is no declaration or if you want to write a test case from scratch against such an object, Silk Test Classic requires a special syntax to uniquely identify the GUI object because there is no identifier.

This special syntax is called a dynamic instantiation and is composed of the class and tag of the object. The general syntax of this kind of identifier is:

```
class("tag").class("tag"). . . .
```

Example

If there is not a declaration for the **Find** dialog box of the **Notepad** application, the syntax required to identify the object with the Classic Agent looks like the following:

```
MainWin("Untitled - Notepad| $C:\Windows\notepad.exe").DialogBox("Find")
```

To create the dynamic tag, the recorder uses the multiple-tag settings that are stored in the **Record Window Declarations** dialog box. In the example shown above, the tag for the **Notepad** contains its caption as well as its window ID.

For the Open Agent, the syntax for the same example looks like the following:

```
FindMainWin("/MainWin[@caption='Untitled - Notepad']").FindDialogBox("Find")
```

Overview of Application States

When testing an application, typically, you have a number of test cases that have identical setup steps. Rather than record the same steps over and over again, you can record the steps as an application state and then associate the application state with the relevant test cases.

An application state is the state you want your application to be in after the base state is restored but before you run one or more test cases. By creating an application state, you are creating reusable code that saves space and time. Furthermore, if you need to modify the Setup stage, you can change it once, in the application state routine.

At most, a test case can have one application state associated with it. However, that application state may itself be based on another previously defined application state. For example, assume that:

- The test case Find is associated with the application state Setup.
- The application state Setup is based on the application state OpenFile.
- The application state OpenFile is based on the built-in application state, DefaultBaseState.
- Silk Test Classic would execute the programs in this order:
 1. DefaultBaseState application state.
 2. OpenFile application state.
 3. Setup application state.
 4. Find test case.

If a test case is based on a single application state, that application state must itself be based on DefaultBaseState in order for the test case to use the recovery system. Similarly, if a test case is based on a chain of application states, the final link in the chain must be DefaultBaseState. In this way, the built-in recovery system of Silk Test Classic is still able to restore the application to its base state when necessary.

Behavior of an Application State Based on NONE

If an application state is based on the keyword NONE, Silk Test Classic executes the application state twice: when the test case with which it is associated is entered and when the test case is exited.

On the other hand, if an application state is based on DefaultBaseState, Silk Test Classic executes the application state only when the associated test case is entered.

The following example code defines the application state InvokeFind as based on the NONE keyword and associates that application state with the test case TestFind.

```
Appstate InvokeFind () basedon none
  xFind.Invoke ()
  print ("hello")

testcase TestFind () appstate InvokeFind
  print ("In TestFind")
  xFind.Exit.Click ()
```

When you run the test case in Silk Test Classic, in addition to opening the **Find** dialog box, closing it, and reopening it, the test case also prints:

```
hello
In TestFind
hello
```

The test case prints hello twice because Silk Test Classic executes the application state both as the test case is entered and as it is exited.

Example: A Feature of a Word Processor

For purposes of illustration, this topic develops test requirements for the searching feature of the sample Text Editor application using the **Find** dialog box. This topic contains the following:

- Determining what data the feature expects.
- Creating meaningful data combinations.
- Overview of recording the stages of a test case.

When a user enters the criteria for the search and clicks **Find Next**, the search feature attempts to locate the string. If the string is found, it is selected (highlighted). Otherwise, an informational message is displayed.

Determining what data the feature expects

A user can enter three pieces of information in the **Find** dialog box:

- The search can be case sensitive or insensitive, depending on whether the **Case Sensitive** check box is checked or unchecked.
- The search can be forward or backward, depending on whether the **Down** or **Up** option button is clicked.
- The search can be for any combination of characters, depending on the value entered in the **Find What** text box.

Creating meaningful data combinations

To organize this information, it is helpful to construct a table that lists the possible combinations of inputs. From this list, you can then decide which combinations are meaningful and should be tested. A partial table for the **Find** dialog box is shown below:

Case Sensitive	Direction	Search String
Yes	Down	Character
Yes	Down	Partial word (start)
Yes	Down	Partial word (end)
Yes	Down	Word
Yes	Down	Group of words
Yes	Up	Character
Yes	Up	Partial word (start)
Yes	Up	Partial word (end)
Yes	Up	Word
Yes	Up	Group of words

Overview of recording the stages of a test case

A test case performs the included actions in three stages. The following table illustrates these stages, describing in high-level terms the steps for each stage of a sample test case that tests whether the Find facility is working.

Setup

1. Open a new document.
2. Type text into the document.
3. Position the text cursor either before or after the text, depending on the direction of the search.
4. Select **Find** from the **Search** menu.
5. In the **Find** dialog box:
 - Enter the text to search for in the **Find What** text box.
 - Select a direction for the search.
 - Make the search case sensitive or not.
 - Click **Find Next** to do the search.
6. Click **Cancel** to close the **Find** dialog box.

Verify

Record a 4Test verification statement that checks that the actual search string found, if any, is the expected search string.

Cleanup

1. Close the document.
2. Click **No** when prompted to save the file.

After learning the basics of recording, you can record from within a test plan, which makes recording easier by automatically generating the links that connect the test plan to the test case.

Verification

This section describes how you can verify one or more characteristics, or properties, of an object.

Verifying Object Properties

You will perform most of your verifications using properties. When you verify the properties of an object, a `VerifyProperties` method statement is added to your script. The `VerifyProperties` method verifies the selected properties of an object and its children.

Each object has many characteristics, or properties. For example, dialog boxes can have the following verification properties:

- `Caption`
- `Children`
- `DefaultButton`
- `Enabled`
- `Focus`
- `Rect`
- `State`

`Caption` is the text that displays in the title bar of the dialog box. `Children` is a list of all the objects contained in the dialog box, `DefaultButton` is the button that is invoked when you press **Enter**, and so on. In your test cases, you can verify the state of any of these properties.

You can also, in the same test case, verify properties of children of the selected object. For example, the child objects in the **Find** dialog box, such as the text box **FindWhat** and the check box **CaseSensitive**, will also be selected for verification.

By recording verification statements for the values of one or more of an object's properties, you can determine whether the state of the application is correct or in error when you run your test cases.

Verifying Object Properties (Open Agent)

This functionality is supported only if you are using the Open Agent.

Record verification statements to verify the properties of an object.

1. Record a test case.
For information on recording a test case, see *Recording Test Cases With the Open Agent*.
2. While recording, hover the cursor over the object, for which you want to verify a property, and click **Ctrl+Alt**. The **Verify Properties** dialog box opens.
3. Select the properties that you want to verify, by checking the check boxes next to the property names.
To verify all or most properties, click **Select All** and then uncheck individual check boxes.
4. Click **OK** to close the **Verify Properties** dialog box.

When you finish recording the test case and paste the recorded test to the editor, all verifications are also pasted to the test script.

Adding a Verification to a Script while Recording

This functionality is supported only if you are using the Open Agent.

Do the following to add a verification to a script during recording:

1. Begin recording.

2. Move the mouse cursor over the object that you want to verify and press **Ctrl+Alt**.

When you are recording a mobile Web application, you can also click on the object and click **Add Verification**.



Note: For any application that uses `Ctrl+Shift` as the shortcut key combination, press **Ctrl+Shift**.

This option temporarily suspends recording and displays the **Verify Properties** dialog box.

3. To select the property that you want to verify, check the corresponding check box.
4. Click **OK**. Silk Test Classic adds the verification to the recorded script and you can continue recording.

Overview of Verifying Bitmaps

A bitmap is a picture of some portion of your application. Verifying a bitmap is usually only useful when the actual appearance of an object needs to be verified to validate application correctness. For example, if you are testing a drawing or CAD/CAM package, a test case might produce an illustration in a drawing region that you want to compare to a baseline. Other possibilities include the verification of fonts, color charts, and certain custom objects.

When comparing bitmaps, keep the following in mind:

- Bitmaps are not portable between GUIs. The format of a bitmap on a PC platform is `.bmp`.
- A bitmap comparison will fail if the image being verified does not have the same screen resolution, color, window frame width, and window position when the test case is run on a different machine than the one on which the baseline image was captured.
- Make sure that your test case sets the size of the application window to the same size it was when the baseline bitmap was captured.
- Capture the smallest possible region of the image so that your test is comparing only what is relevant.
- If practical, do not include the window's frame (border), since this may have different colors and/or fonts in different environments.

Verifying Appearance Using a Bitmap

When you are using the Classic Agent, use this procedure to compare the actual appearance of an image against a baseline image. Or, use it to verify fonts, color charts, or custom objects.



Note: To verify a bitmap when you are using the Open Agent, you can add the `VerifyBitmap` method to your script. The `VerifyBitmap` method is supported for both agents.

1. Complete the steps in *Verifying a Test Case*.
2. On the **Verify Window** dialog box, click the **Bitmap** tab and then select the region to update: **Entire Window**, **Client Area of Window** (that is, without scroll bar or title bar), or **Portion of Window**.

3. In the **Bitmap File Name** text box, type the full path of the bitmap file that will be created.

The default path is based on the current directory. The default file name for the first bitmap is `bitmap.bmp`. Click **Browse** if you need help choosing a new path or name.

4. Click **OK**. If you selected **Entire Window** or **Client Area of Window**, Silk Test Classic captures the bitmap and returns you to your test application. If you selected **Portion of Window**, position the cursor at the desired location to begin capturing a bitmap. While you press and hold the mouse button, drag the mouse to the screen location where you want to end the capture. Release the mouse button.

A bitmap comparison will fail if the image being verified does not have the same screen resolution, color, window frame width, and window position as the baseline image.

Capture the smallest possible region of the image so that your test is comparing only what is relevant.

5. If you are writing a complete test case, record the cleanup stage and paste the test case into the script. If you have added a verification statement to an existing test case, paste it into your script and close the **Record Actions** dialog box.

Overview of Verifying an Objects State

Each class has a set of methods associated with it, including built-in verification methods. You can verify an object's state using one of these built-in verification methods or by using other methods in combination with the built-in `Verify` function.

A class's verification methods always begin with `Verify`. For example, a `TextField` has the following verification methods; `VerifyPosition`, `VerifySelRange`, `VerifySelText`, and `VerifyValue`.

You can use the built-in `Verify` function to verify that two values are equal and generate an exception if they are not. Typically, you use the `Verify` function to test something that does not map directly to a built-in property or method. `Verify` has the following syntax:

```
Verify (aActual, aExpected [, sDesc])
```

aActual	The value to verify. ANYTYPE.
aExpected	The expected value. ANYTYPE.
sDesc	<i>Optional:</i> A message describing the comparison. STRING.

Usually, the value to verify is obtained by calling a method for the object being verified; you can use any method that returns a value.

Example: Verify an object

This example describes how you can verify the number of option buttons in the **Direction RadioList** in the **Replace** dialog box of the Text Editor. There is no property or method you can directly use to verify this. But there is a method for **RadioList**, `GetItemCount`, which returns the number of option buttons. You can use the method to provide the actual value, then specify the expected value in the script.

When doing the verification, position the mouse pointer over the **RadioList** and press `Ctrl+Alt`. Click the **Method** tab in the **Verify Window** dialog box, and select the `GetItemCount` method.

Click **OK** to close the **Verify Window** dialog box, and complete your test case. Paste it into a script. You now have the following script:

```
testcase VerifyFuncTest ()
  TextEditor.Search.Replace.Pick ()
  Replace.Direction.GetItemCount ()
  Replace.Cancel.Click ()
```

Now use the `Verify` function to complete the verification statement. Change the line:

```
Replace.Direction.GetItemCount ()
```

to

```
Verify (Replace.Direction.GetItemCount (), 2)
```

That is, the call to `GetItemCount` (which returns the number of option buttons) becomes the first argument to `Verify`. The expected value, in this case, 2, becomes the second argument.

Your completed script is:

```
testcase VerifyFuncTest ()
  TextEditor.Search.Replace.Pick ()
  Verify (Replace.Direction.GetItemCount (), 2)
  Replace.Cancel.Click ()
```

Fuzzy Verification

There are situations when Silk Test Classic cannot see the full contents of a control, such as a text box, because of the way that the application paints the control on the screen. For example, consider a text box whose contents are wider than the display area. In some situations the application clips the text to fit the display area before drawing it, meaning that Silk Test Classic only sees the contents that are visible; not the entire contents.

Consequently, when you later do a `VerifyProperties` against this text box, it may fail inappropriately. For example, the true contents of the text box might be `29 Pagoda Street`, but only `29 Pagoda` displays. Depending on how exactly the test is created and run, the expected value might be `29 Pagoda` whereas the value seen at runtime might be `29 Pagoda Street`, or vice versa. So the test would fail, even though it should pass.

To work around this problem, you can use fuzzy verification, where the rules for when two strings match are loosened. Using fuzzy verification, the expected and actual values do not have to exactly match. The two values are considered to match when one of them is the same as the first or last part of the other one. Specifically, `VerifyProperties` with fuzzy verification will pass whenever any of the following functions would return `TRUE`, where `actual` is the actual value and `expected` is the expected value:

- `MatchStr (actual + "*", expected)`
- `MatchStr ("*" + actual, expected)`
- `MatchStr (actual, expected + "*")`
- `MatchStr (actual, "*" + expected)`

In string comparisons, `*` stands for any zero or more characters.

For example, all the following would pass if fuzzy verification is enabled:

Actual Value	Expected Value
29 Pagoda	29 Pagoda Street
oda Street	29 Pagoda Street
29 Pagoda Street	29 Pagoda
29 Pagoda Street	oda Street

Enabling fuzzy verification

You enable fuzzy verification by using an optional second argument to `VerifyProperties`, which has this prototype:

```
VerifyProperties (WINPROPTREE WinPropTree [,FUZZYVERIFY FuzzyVerifyWhich])
```

where the `FUZZYVERIFY` data type is defined as:

```
type FUZZYVERIFY is BOOLEAN, DATACLASS, LIST OF DATACLASS
```

So, for the optional `FuzzyVerifyWhich` argument you can either specify `TRUE` or `FALSE`, one class name, or a list of class names.

FuzzyVerifyWhich value

FALSE (default) Fuzzy verification is disabled.

One class Fuzzy verification is enabled for all objects of that class.

Example `window.VerifyProperties ({...}, Table)` enables fuzzy verification for all tables in window (but no other object).

List of classes

Fuzzy verification is enabled for all objects of each listed class.

Example `window.VerifyProperties ({...}, {Table, TextField})` enables fuzzy verification for all tables and text boxes in window (but no other object).

TRUE

Fuzzy verification is enabled only for those objects whose `FuzzyVerifyProperties` member is `TRUE`.

To set the `FuzzyVerifyProperties` member for an object, add the following line within the object's declaration:

```
FUZZYVERIFY FuzzyVerifyProperties = TRUE
```

Example: If in the application's include file, the `DeptDetails` table has its `FuzzyVerifyProperties` member set to `TRUE`:

```
window ChildWin EmpData
. . .
    Table DeptDetails
        FUZZYVERIFY FuzzyVerifyProperties = TRUE
```

And the test has this line:

```
EmpData.VerifyProperties ({...}, TRUE)
```

Then fuzzy verification is enabled for the `DeptDetails` table (and other objects in `EmpData` that have `FuzzyVerifyProperties` set to `TRUE`), but no other object.

Fuzzy verification takes more time than standard verification, so only use it when necessary.

For more information, see the `VerifyProperties` method.

Defining your own verification properties

You can also define your own verification properties.

Verifying that a Window or Control is No Longer Displayed

1. Click **Record > Testcase** to begin recording a test case and drive your application to the state you want to verify. To record a verification statement in an existing test case, click **Record > Actions**.
2. When you are ready to record a verification statement, position the mouse cursor over the object you want to verify, and press `Ctrl+Alt`. Silk Test Classic displays the **Verify Window** dialog box over your application window.
3. Click the **Property** tab. Silk Test Classic lists the properties for the selected window or control on the right.
4. Make sure that only the `Exists` property is selected for the window or control.
If additional properties are selected, the verification will fail because the actual list of properties will differ from the expected list.
5. Change the value in the **Property Value** field from `TRUE` to `FALSE`.
6. Click **OK** to accept the `Exists` property for the selected window or control. Silk Test Classic closes the **Verify Window** dialog box and displays the **Record Status** window. The test case will verify that the window or control has the property value of `FALSE`, verifying that the object is no longer displayed. If not, Silk Test Classic writes an error to the results file.

Data-Driven Test Cases

Data-driven test cases enable you to invoke the same test case multiple times, once for each data combination stored in a data source. The data is passed to the test case as a parameter. You can think of a data-driven test case as a template for a class of test cases. Data-driven test cases offer the following benefits:

- They reduce redundancy in a test plan.
- Writing a single test case for a group of similar test cases makes it easier to maintain scripts.
- They are reusable; adding new tests only requires adding new data.

Regardless of the technique you use, the basic process for creating a data-driven test case is:


1. Create a standard test case. It will be very helpful to have a good idea of what you are going to test and how to perform the verification.
2. Identify the data in the test case and the 4Test data types needed to store this data.
3. Modify the test case to use variables instead of hard data.
4. Modify the test case to specify input arguments to be used to pass in the data. Replace the hard coded data in the test case with variables.
5. Call the test case and pass in the data, using one of four different techniques:
 - Use a database and the **Data Driven Workflow** to run the test case. Micro Focus recommends using this method for data-driven testing.
 - Click **Run > Testcase** and type the data into the **Run Testcase** dialog box.
 - In a QA Organizer test plan, insert the data as an attribute to a test description.
 - If the data exists in an external file, write a function to read the file and use a `main()` function to run the test case.

Data-Driven Workflow

You can use the **Data Driven Workflow** to create data-driven test cases that use data stored in databases. The **Data Driven Workflow** generates much of the necessary code and guides you through the process of creating a data-driven test case.

Before you can create and run data-driven test cases, you need to perform the following actions:

1. Record a standard test case.
2. Set up or identify the existing data source with the information you want to use to run the test.
3. Configure your Data Source Name (DSN), if you are not using the default, which is *Silk DDA Excel*.

 **Note:** When you use the **Data Driven Workflow**, Silk Test Classic uses a well-defined record format. To run data-driven test cases that were not created through the **Data Driven Workflow**, you need to convert your recordings to the new record format. To run data-driven test cases that do not follow the record format, run the tests outside of the **Data Driven Workflow**.

To enable or disable the **Data Driven Workflow**, click **Workflows > Data Driven**.



To create and execute a data-driven test case, sequentially click each icon in the workflow bar to perform the corresponding procedure.

Action	Description
Data Drive Testcase	Select a test case to data drive. Silk Test Classic copies the selected test case and creates a new data-driven test case by adding a "DD_" prefix to the original name of the test case. Silk Test Classic also writes other data-driven information to the new or existing data driven script file (.g.t file).
Find/Replace Values	Find and replace values in the new test case with links to the data source.
Run Testcase	Run the data-driven test case, optionally selecting the rows and tables in the data source that you want to use.
Explore Results	View test results.

Working with Data-Driven Test Cases

Consider the following when you are working with data-driven test cases:

- The **4Test Editor** contains additional menu selections and toolbars for you to use.
- Silk Test Classic can data drive only one test case at a time.
- You cannot duplicate test case names. Data-driven test cases in the same script must have unique names.
- The Classic 4Test editor is not available with data-driven test cases in .g.t files.
- You cannot create data-driven test cases from test cases in .inc files; you can only create data-driven test cases from test cases in .t or .g.t files. However, you can open a project, add the *.inc, select the test case from the test case folder of the project, and then select data drive.
- When you data drive a [use '<script>.t'] is added to the data-driven test case. This is the link to the .t file where the test case originated. If you add a test case from another script file then another use line pointing to that file is added. If the script file is in the same directory as the <script.g.t>, then no path is given, otherwise, the absolute path is added to the use line. If this path changes, it is up to you to correct the path; Silk Test Classic will not automatically update the path.
- When you open a .g.t file using **File > Open**, Silk Test Classic automatically loads the data source information for that file. If you are in a .g.t file and that file's data source is edited, click **Edit > Data Driven > Reload Database** to refresh the information from the data source.
- If you add a new data-driven test case to an existing .g.t file that is fully collapsed, Silk Test Classic expands the previous test case, but does not edit it.

Code Automatically Generated by Silk Test Classic

When you create a data-driven test case, Silk Test Classic verifies that the DSN configuration is correct by connecting to the database, generates the 4Test code describing the DSN, and writes that information into the data-driven script.

Do not delete or change the information created by Silk Test Classic. If you do, you may not be able to run your data-driven test case.

When you click **OK** on the **Specify Data Driven Testcase** dialog box, Silk Test Classic automatically writes the following information to the top of your data driven script file.

The information is delivered "rolled up" (collapsed); in order to see the details you need to click on the plus sign to expand the code:

```
[+] // *** DATA DRIVEN ASSISTANT Section (!! DO NOT REMOVE !!) ***
```

The .inc files used by the original test cases, and the .t file indicating where the test case just came from, in this case from Usability.t:

```
[ ] use "datadrivetc.inc"
[ ] use "Usability.t"
```

A reference to the DSN, specifying the connect string, including username and password, for example:

```
[ ] // *** DSN ***
[ ] STRING gsDSNConnect = "DSN=SILK DDA Excel;DBQ=C:\ddatesting
\TestExcel.xls;UID=;PWD=;"
```

Each data-driven test case takes as a single argument a record consisting of a record for each table that is used in the test case. The record definition is automatically generated as shown here:

```
[+] // testcase VerifyProductDetails (REC_DATALIST_VerifyProductDetails rdVpd)
[ ] // Name: REC_<Testcase name>. Fields Types: Table record types. Field
Names: Table record
type with 'REC_' replaced by 'rec'
[-] type REC_DATALIST_VerifyProductDetails is record
[ ] REC_Products recProducts
[ ] REC_Customers recCustomers
[ ] REC_CreditCards recCreditCards
```

Each table record contains the column names in the same order as in the database. Spaces in table and column names are removed. Special characters such as \$ are replaced by underscores.

```
[ ] // *** Global record for each Table ***
[ ]
[-] type REC_Products_ is record
[ ] STRING Item //Item,
[ ] REAL Index //Index,
[ ] STRING Name //Name,
[ ] REAL ItemNum //ItemNum,
[ ] STRING Price //Price,
[ ] STRING Desc //Desc,
[ ] STRING Blurb //Blurb,
[ ] REAL NumInStock //NumInStock,
[ ] INTEGER QtyToOrder //QtyToOrder,
[ ] INTEGER OnSale //OnSale,
```

Silk Test Classic writes a sample record for each table. This is the data used if you opt to use sample data on the **Run Testcase** dialog box. A value from the original test case is inserted into the sample record, even if there are syntax errors when that column is first used to replace a value.

```
[ ] // *** Global record containing sample data for each table ***
[ ] // *** Used when running a testcase with 'Use Sample Data from Script'
checked ***
[ ]
[-] REC_Products_ grTest_Products_ = {...}
[ ] NULL // Item
[ ] NULL // Index
[ ] NULL // Name
[ ] NULL // ItemNum
[ ] NULL // Price
[ ] NULL // Desc
[ ] NULL // Blurb
[ ] NULL // NumInStock
[ ] 2 // QtyToOrder
[ ] NULL // OnSale
[ ]
[ ] // *** End of DATA DRIVEN ASSISTANT Section ***
```

Tips And Tricks for Data-Driven Test Cases

There are several things to know about working with data sources while you are creating data-driven test cases.

- You must have an existing data source with tables and columns defined before you data drive a test case. However, the data source does not need to contain rows of data. You cannot use the *Data Driven Workflow* to create data sources or databases.

- If you have a table in your data source that has a long name (greater than 25 characters), all of the name may not be visible in the **Find and Replace** menu bar in the **4Test Editor**. You may find it helpful to change the size of the menu bar to display more of your table name.
- You cannot change to a different data source once you have started to find and replace values in a script. If you do, you will have problems with prior replacements. If you want to change your data source, you should create a new data-driven script file.
- If you are working with a data source that requires a user name and password, you can add the username and password to the connect string in the `.g.t` file. The first example below shows how SQL Server requires a userid and password. [] `STRING gsDSNConnect = "DSN=USER.SQL.DSN;UID=SA;PWD=sesame;"` where `UID=<your user ID>` ("SA" in the example above) and where `PWD=<your password>` ("sesame" in the example above). On the other hand, the example below shows how the Connect string for a MS Excel DSN does not require user IDs or passwords: [] `STRING gsDSNConnect = "DSN=Silk DDA Excel;DBQ=C:\TestExcel.xls;UID=;PWD="`
- You can choose to run with a sample record if the table is empty; however, this record is not inserted into the database. The sample record is created by Silk Test Classic when it replaces values from the test case by the table and columns in your database.
- Real numbers should be stored as valid 4Test Real numbers with format: `[-]ddd.ddd[e[-]ddd]`, even though databases such as MS Excel allow a wider range of formats – for example, currencies and fractions.
- There are no restrictions on how you name your tables and columns within your data source. Silk Test Classic automatically removes spaces, and converts dollar signs and other special characters to underscores when it creates the sample record and writes other code to your data-driven test case. Silk Test Classic handles MS Excel and MS Access table names without putting quotation marks around them. This means that your table and column names will look familiar when you go to find and replace values.
- If you encounter the error "ODBC Excel Driver numeric field overflow" while running a test case, check the Excel workbook that you are using as your data source. You may have some columns that are defined as `STRING` columns but contain numeric values in some of the rows. If you have a column that you want to treat as numeric strings rather than as numbers, either format the column as 'Text' or begin the number strings with a single-quote character. For example: '1003 instead of: 1003
- If modifying data sources in an existing Excel data sheet, use the **remove column** option to delete any data to be removed, as simply deleting from the cell, using clear contents, or copy/pasting content will not register correctly with the DDS file in Silk Test Classic and may lead to a data source mismatch error: `*** Error: Incompatible types -- Number of list elements exceeds number of fields.`

Formatting MS Excel worksheets for use as a data source

Use the 'General' format for the columns of your worksheets. Here are specific suggestions for column formats based on the intended data type of the column:

Intended Data Type of Column	Excel Column Format
STRING	If the column contains only text, no numbers, dates or booleans, then apply the 'General' format. If the column contains text and numbers, then you can still apply the 'General' format if you begin the number strings with a single-quote character. For example: '1003 instead of: 1003. Otherwise, apply the 'Text' format.
INTEGER or REAL	'General' or 'Number' format.
BOOLEAN	'General' format. Use only the values TRUE and FALSE.

Intended Data Type of Column	Excel Column Format
DATETIME	'Custom' format: yyyy-mm-dd hh:mm:ss. That agrees with the ISO format used by Silk Test Classic DATETIME values.

Testing an Application with Invalid Data

This topic assumes that you are familiar with data driving test cases.

To thoroughly test an application feature, you need to test the feature with invalid as well as valid data.

For example, the sample Text Editor application displays a message box if a user specifies a search string in the **Find** dialog box that doesn't exist in the document. To account for this, you can create a data-driven test case, like the following, that verifies that the message box displays and has the correct message:

```
type SEARCHINFO is record
  STRING sText      // Text to type in document window
  STRING sPos       // Starting position of search
  STRING sPattern   // String to look for
  BOOLEAN bCase     // Case-sensitive or not
  STRING sDirection // Direction of search
  STRING sExpected  // The expected match
  STRING sMessage   // The expected message in message box

testcase FindInvalidData (SEARCHINFO Data)
  TextEditor.File.New.Pick ()
  DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
  TextEditor.Search.Find.Pick ()
  Find.FindWhat.SetText (Data.sPattern)
  Find.CaseSensitive.SetState (Data.bCase)
  Find.Direction.Select (Data.sDirection)
  Find.FindNext.Click ()

  MessageBox.Message.VerifyValue (Data.sMessage)
  MessageBox.OK.Click ()

  Find.Cancel.Click ()
  TextEditor.File.Close.Pick ()
  MessageBox.No.Click ()
```

The `VerifyValue` method call in this test case verifies that the message box contains the correct string. For example, the message should be `Cannot find Ca` if the user enters `Ca` into the **Find** dialog box and the document editing area does not contain this string.

Enabling and Disabling Workflow Bars

Only one workflow bar can be enabled at a time.

To enable or disable a workflow bar, click **Workflows** and then select the workflow bar that you want to turn on or off. For example, click **Workflows > Basic**.

You can select one of the following workflows:

Workflow	Description
Basic workflow	Guides you through the process of creating a test case.
Data Driven workflow	Guides you through the process of creating a data-driven test case.

Data Source for Data-Driven Test Cases

When you install Silk Test Classic, the `SILK DDA EXCEL` DSN is copied to your installation computer. This is the default DSN that Silk Test Classic uses. This DSN uses a MS Excel 8.0 driver and does not have a particular workbook (`.xls` file) associated with it.

The **Select Data Source** dialog box allows you to choose the data source:

- For new data-driven test cases, choose *Silk DDA Excel*.
- For backward compatibility, choose *Segue DDA Excel*. This allows existing `.g.t` files that reference *Segue DDA Excel* to continue to run.

You do not have to use the default DSN. For additional information when using a different DSN, see *Configuring Your DSN*.

You may use any of the following types of data sources:

- Text files and comma separated value files (`*.txt` and `*.csv` files)
- Microsoft Excel
- Microsoft SQL Server
- Microsoft Access
- Oracle
- Sybase SQL Anywhere

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Configuring Your DSN

The default DSN for data-driven test cases, *Silk DDA Excel*, is created during the installation of Silk Test Classic. To use the default DSN you do not need to configure your DSN.

The **Select Data Source** dialog box allows you to choose the data source:

- For new data-driven test cases, choose *Silk DDA Excel*.
- For backward compatibility, choose *Segue DDA Excel*. This allows existing `.g.t` files that reference *Segue DDA Excel* to continue to run.

The following instructions show how to configure a machine to use a different DSN than the *Silk DDA Excel* default.

1. Click **Start > Control Panel > System and Security > Administrative Tools > Data Sources (ODBC)**.
2. On the **ODBC Data Source Administrator**, click either the **System DSN** tab or the **User DSN** tab, depending on whether you want to configure this DSN for one user or for every user on this machine.
3. Click **Add**.
4. On the **Create New Data Source** dialog box, select the driver for the data source and click **Finish**.
To restore the default DSN for Silk Test Classic, select the driver for Microsoft Excel Driver (`*.xls`).
5. On the setup dialog box of the data source, enter a name for the data source.
To restore the default for Silk Test Classic, enter `SILK DDA EXCEL`. For additional information about the dialog box, refer to the database documentation or contact your database administrator.
6. Click **OK**.

Setting Up a Data Source

Before you can run a data-driven test case you must set up a file that contains the tables, which are called *worksheets* in Microsoft Excel (Excel), and the columns that you want to use. The tables do not have to be populated with data, but it might help to have at least one complete record filled out.

1. Open one of the data sources for data-driven test cases, for example Excel.
2. Name at least one table, or worksheet if you are using Excel, and create column names for the table.
3. Save the data source.

Example

The Excel file `TestExcel.xls` can be used as a data source for a data-driven test case and includes the three worksheets *Products*, *Customers*, and *CreditCards*. The *Customers* worksheet includes the columns *Customer*, *Name*, *Address*, and so on.

	A	B	C	D	E	F	
1	Customer	Name	Address	City	State	Zip	P
2	Cust 1	Test Name1	123 Street1	City1	MA	02421	781-111
3	Cust 2	Test Name2	123 Street2	City2	MA	02422	781-222
4	Cust 3	Test Name3	123 Street3	City3	MA	02421	783-333
5	Cust 4	Test Name4	123 Street4	City4	MA	02422	784-444
6	Cust 5	Test Name5	123 Street5	City5	MA	02421	785-555
7	Cust 6	Test Name6	123 Street6	City6	MA	02422	786-666
8							
9							

Using an Oracle DSN to Data Drive a Test Case

To use an Oracle DSN to data drive a test case, select the test case to data drive, let Silk Test Classic generate code into the new test case file, and then make the following manual modifications to the DSN:

1. Find out which columns are included in the table of your schema.
Different schemas may contain tables with the same name. The table lists for the **Find/Replace Values** dialog box, the re-sizable menu bar, and the **Specify Rows** dialog box will list the same table name once for each schema without indicating the schema. For each of those list items the column list will contain the names of the columns in all of the tables with that name.
2. After finding and replacing values, split each table record into separate records according to the schema. Do that for the sample record as well.
The record names should have the form: `<Record prefix><schema>_<table>`. For example, if the schema is `STUser` and the table is `Customers`, the name of the table record type will be `REC_STUser_Customers` and the declaration for the field in the test case record for the table will be `REC_STUser_Customers recSTUser_Customers // Customers`.
3. Run the test case from a test plan, unless you are running all rows for all tables. Use the **Specify Rows** dialog box to build the `ddatestdata` value, then modify that value to include the schema name in the query.



Note: Specify a query for every table, even if you want to run all rows for a table. To run all rows, leave the where clause blank.

Creating the Data-Driven Test Case

This section describes how you can create a data-driven test case.

Selecting a Test Case to Data Drive

For information on the steps that you need to complete before you can select a test case to data drive, see *Data-Driven Workflow*.

While you are in a script, choose one of the following to select a test case for data driving:

- Click **Tools > Data Drive Testcase**.
- Right-click into the script and select **Data Drive Testcase**.

When you select a test case, Silk Test Classic copies the selected test case and creates a new data-driven test case by adding a `DD_` prefix to the original name of the test case. Silk Test Classic also writes other data-driven information to the new or existing data-driven script file `script.g.t`.

Finding and Replacing Values

For information on the steps that you need to complete before you can find and replace values in a test case, see *Data-Driven Workflow*.

You can use the **Find/Replace Values** dialog box to find and replace values in a data-driven test case with links to values in tables and columns. Values are text strings, numbers, and booleans that exist in your original test cases. One of the steps in creating a data-driven test case is to find these values and replace them with references to columns in your data source.

Silk Test Classic checks to make sure that each value you select is appropriate for replacement by the column in your test case. You can turn off this validation by clicking **Edit > Data Driven > Validate Replacements** while you are in a `.g.t` file. This means that the **Find** aspect of **Find and Replace** works as usual, but that the values that you replace are not validated. By turning off this checking, you suppress the error messages that Silk Test Classic would have otherwise displayed. Any 4Test identifier or fragment of a string is considered an invalid value for replacement unless **Validate Replacements** is turned off. If you are new to creating data-driven test cases, we recommend that you keep this validation turned on.

 **Note:** You can also use the **Find and Replace** menu bar in the **4Test Editor** to find and replace values in a data-driven test case.

When you are using **Find and Replace**, sometimes a method requires a data type that does not match the column that you want to replace. For example, `SetText` requires a string, but you may want to set a number instead, or perhaps the database does not store data in the 4Test type that you would like to use. Silk Test Classic can handle these kinds of conversions, with a few exceptions.

1. Open the **Find/Replace Values** dialog box.
 - Right-click into a test case in a `.t` file and select **Data Driven Testcase**. Specify the data source, the data-driven script, and the data-driven test case. When you complete the **Specify Data Driven Testcase** dialog box and the data-driven script opens in the **4Test Editor**, the **Find/Replace Values** dialog box opens automatically.
 - Highlight a value in a `.g.t` file and choose **Edit > Data Driven > Find/Replace Values**.
 - Right-click a value in a `.g.t` file and select **Find > Replace Values**.
2. Check the **Text**, **Number**, and **True/False** check boxes to specify what type of values you want to find or replace.

By default all check boxes are checked, indicating that Silk Test Classic will look for text (string), number, and true/false (boolean) values in the test case.
3. Select **Up** or **Down** to define the direction in which Silk Test Classic searches in the file for the value. The default is **Down**.
4. Select the name of the table that contains the values that you want to link to from the **Table** list.
5. Select the name of the column that contains the values that you want to link to from the **Column** list.
6. Click **Find Next** to start searching for the values within the test case. Silk Test Classic finds and highlights various values.

7. Click **Replace** to replace the found value with a link to the values in the data source.
8. When you have finished replacing values, click **Cancel**.

Check the **Don't show me this again** check box if you want to use the re-sizable menu bars instead of the **Find/Replace Values** dialog box to find and replace values. You can re-display the dialog box by clicking **Edit > Data Driven > Find/Replace Values** or by right-clicking into a data-driven test case and unchecking **Validate Replacements**.

Rules and Tips for Finding and Replacing Values in the Data-Driven Workflow

Rules for finding and replacing

The following table shows how Silk Test Classic works when it finds and replaces values in your scripts.

Value Found in Script	Database Column Type	Replaced by	Tip
STRING	STRING	Column	
	INTEGER	Str(Column)	5
	REAL	Str(Column,NULL,2)	1, 3, 4, 5
	BOOLEAN	Str(Column)	8
	DATETIME	[STRING]Column	7
INTEGER	STRING	Val(Column)	
	INTEGER	Column	
	REAL	Column	1, 6
	BOOLEAN	(not allowed)	
	DATETIME	(not allowed)	
REAL	STRING	Val(Column)	
	INTEGER	Column	6
	REAL	Column	
	BOOLEAN	(not allowed)	
	DATETIME	(not allowed)	
BOOLEAN	STRING	(not allowed)	
	INTEGER	Column	2, 6
	REAL	(not allowed)	
	BOOLEAN	Column	
	DATETIME	(not allowed)	
DATETIME			7

Tips for finding and replacing values

When you are creating a data-driven test case, there are several points to note about the find and replace process. These notes are referenced by the Rules for finding and replacing table, shown above.

- MS Excel stores integers as REALs. This means that you may see a 1 in an MS Excel cell, but the number that is stored is actually 1.000. Whenever you replace an integer in a test case with a column

designated as REAL, Silk Test Classic displays a question similar to the following: *To ensure that Silk Test Classic handles your data properly, additional information is needed: Will column QtyToOrder contain only INTEGER (no decimal point) values? Click Yes to treat that column's values as INTEGER; click No to continue to treat that column's values as REAL.*

- Some databases (such as MS Excel) store booleans as INTEGERS. This means that you may see a TRUE in an Excel cell, but the value that is stored is actually 1. Whenever you replace a boolean in a test case with a column designated as INTEGER, Silk Test Classic displays a question similar to the following: *To ensure that Silk Test Classic handles your data properly, additional information is needed: Will column OnSale contain only BOOLEAN (TRUE/FALSE) values? Click Yes to treat that column's values as BOOLEAN; click No to continue to treat that column's values as INTEGER.*
- If you have a currency string, we recommend that you split it into two strings separated by a plus sign. The first string should contain just the financial symbol (for example, the dollar sign or USD acronym) and the second string should contain just the number value. When you do a find and replace, this lets you replace only the number value, not the financial symbol or acronym. For example, do not use \$123.45, instead use \$ + 123.45

- When a STRING literal is replaced by a column containing REAL values, the real number is rounded to two decimal places. To change the number of decimal places that are displayed, change the value of iDecimal in Str (table.column, NULL, iDecimal). For example, to display 7.1234 instead of 7.12, change:

```
Str (table.column, NULL, 2)
```

to:

```
Str (table.column, NULL, 4)
```

- You can replace any STRING value with a REAL or INTEGER column. However, in order to have a working sample record, the value that is first replaced by that column should be a STRING representation of a REAL or INTEGER constant.
- In certain cases, 4Test implicitly converts one data type to another. In that case, Silk Test Classic does not insert a conversion expression when replacing the value by the DB column reference..
- To represent a DATETIME value, in other words a 4Test DATETIME constant, you must create a STRING that specifies the date and time in the standard 4Test DATETIME (ISO) format - YYYY-MM-DD HH:MM:SS.MSMSMS. You can truncate the string at any point, as long as the last field is complete. If you use a different format, you will cause a runtime error. Because DATETIME values found in a script are actually STRING values, the replacement rules are the rules for STRING values.
- The conversion from STRING to BOOLEAN will cause a runtime error. The expression must be changed manually from 'Str(Column)' to: [STRING]Column.
- A fraction is considered an INTEGER division expression, so the value inserted into the sample record is truncated to the next lowest integer.
- If Silk Test Classic does not insert any of the expected conversion expressions, then the **Validate Replacements** setting is probably unchecked. Click **Edit > Data Driven > Validate Replacements** to see how **Validate Replacements** is set.
- When you do a Find/Replace, Silk Test Classic skips values contained in variable definitions. To find those values, click **Edit > Data Driven > Validate Replacements** to turn off **Validate Replacements**. We recommend that you initialize variables after you define them; be sure to separate the variable definition and assignment on two different lines.

Running a Data-Driven Test Case

Once you have selected a test case to data drive, and found and replaced values, choose one of the following ways to run the test case:

- Click **Run > Run** while in a .g.t file. This command runs main(), or if there is no main(), the command runs all test cases. For each test case, this command runs all rows for all tables used by the test case.
- Click **Run > Testcase** and select the data-driven test case from the list of test cases on the **Run Testcase** dialog box, for all tables used by the test case.
- Click **Run > Testcase > Run** to run the test case for all rows for all tables used by the test case.

Running a Test Case Using a Sample Record for Each Table Used by the Data-Driven Test Case

This is useful if you want to do a quick test or are not connected to your data source. The sample record is created as you replace values in the test case. When you first use a column to replace a test case value, that value is inserted into the table record in the field for that column.

1. On the **Run Testcase** dialog box, click **Use Sample Data from Script**.

By default, Silk Test Classic runs every combination of rows in your tables. The number of test cases that runs is:

```
# of rows selected for Table 1 X the # of rows selected for  
Table 2 X the number of rows for Table 3  
... and so on
```

For example, if your test case uses 3 tables with 5 rows each, Silk Test Classic will run 125 test cases.

2. To select the rows you want to run on a table-by-table basis, click **Specify Rows** on the **Run Testcase** dialog box to use the **Specify Rows** dialog box to create a query.
3. Specify arguments, if necessary, in the **Arguments** text box. Remember to separate multiple arguments with commas.
4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

- BaseStateExecutionFinished
- Connecting
- Verify
- Exists
- Is
- Get
- Set
- Print
- ForceActiveXEnum
- Wait
- Sleep

5. To view results using the Silk TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.
6. Click **Run**. Silk Test Classic runs the test case and generates a results file.

Passing Data to a Test Case

Once you have defined your data-driven test case, you pass data to it, as follows:

- If you are not using the test plan editor, you pass data from a script's main function.
- If you are using the test plan editor, you embed the data in the test plan and the test plan editor passes the data when you run the test plan.

Example Setup for Forward Case-Sensitive Search

Here is a sample application state that performs the setup for all forward case-sensitive searches in the **Find** dialog box:

```
appstate Setup () basedon DefaultBaseState  
TextEditor.File.New.Pick ()
```

```
DocumentWindow.Document.TypeKeys ("Test Case<Home> ")
TextEditor.Search.Find.Pick ()
Find.CaseSensitive.Check ()
Find.Direction.Select ("Down")
```

Building Queries

Before you define a query to access certain data in a data-driven test case, there are several steps you need to complete. For additional information, see *Using the Data Driven Workflow* for more information.

Respond to the prompts on the **Specify Rows** dialog box to create a query for a table. The following are examples of simple queries:

- To find and run the records of customers whose customer ID number is 1001: (CUSTID = 1001)
- To find and run the records of customers whose names begin with the letters "F" or "G": (CUST_NAME LIKE 'F%') OR (CUSTNAME LIKE 'G%').

See the description of the enter values area in the **Specify Rows** dialog box to see examples of more complex queries.

Adding a Data-Driven Test Case to a Test Plan

You can run a data-driven test case from a test plan as either a data-driven test case or as a regular test case. To distinguish between the two cases, there are two keywords for you to use:

- `ddatestcase` specifies the name of a test case that runs as a data-driven test case.
- `ddatestdata` specifies the list of rows that will be run with the data-driven test case.

If the test case is specified with the keyword `ddatestcase`, it is run as a data-driven test case. Use this keyword only with data-driven test cases.

To specify a data-driven test case in a test plan

- Add keyword `ddatestcase` in front of the test case name.
- Add the keyword `ddatestdata` as a list of queries that specify the particular rows you want the test case to run with. The list of queries is represented as a single `LIST OF STRING` parameter.

Rules for using data-driven keywords

- The `ddatestdata` keyword requires simple select queries. To specify the row you want to run a test case with, use the `ddatestdata` keyword with the format: `select * from <table> where ...`
- The keyword `ddatestcase` cannot be a level above the script file and still work. The script file has to be at the same level or above it.
- A test plan needs to specify a test case using either the keyword `testcase` or the keyword `ddatestcase`. Using both causes a compiler error.
- If the `ddatestdata` keyword is present, then the `ddatestcase` is run using the `ddatestdata` value as the rows to run.
- The default is to run all rows for all tables. The value for `ddatestdata` for this is `ALL_ROWS_FOR_ALL_TABLES`.
- Using the keyword `testdata` in a test item with keyword `ddatestcase` will cause a compiler error.
- If the test case is specified with the keyword `testcase`, then the test case is run as a regular test case and the `testdata` keyword or symbols must be present to specify the value that will be passed as the regular argument. This value must be a record of the type defined for the `ddatestcase`, in other words of type `REC_DATALIST_<Testcase name>`.

You can add a data-driven test case to a test plan by using the **Testplan Detail** dialog box or by editing the test plan directly. However, if you edit the test plan directly, then the keywords are not automatically validated and it is your responsibility to make sure that the keywords, which are `testcase` versus `ddatestcase` and `testdata` versus `ddatestdata`, are appropriate for the intended execution of the test case.

Whenever you use the **Test Detail** dialog box, be sure to click the **Testcases** button and select the test case from the list. That will ensure that the proper keywords are inserted into the test plan.

Using sample records data within test plans

To run a test case with the sample record within a test plan, you must manually input the test data, in the format `ddatestdata: {"USE_SAMPLE_RECORD_<tablename>"}`

For example:

```
script: example.t
ddatestcase: sampletc
ddatestdata: {"USE_SAMPLE_RECORD_SpaceTable$"}
```

You must put the `USE_SAMPLE_RECORD_` prefix in front of each table name that you want to run against. If you are using two tables, you need to input the prefix twice, as shown below with two tables named "Table1" and "Table2":

```
ddatestdata: {"USE_SAMPLE_RECORD_Table1", "USE_SAMPLE_RECORD_Table2" }
```

Using a main Function in the Script

Although most of the script files you create contain only test cases, in some instances you need to add a function named `main` to your script. You can use the `main` function to pass data to test cases as well as control the order in which the test cases in the script are executed.

When you run a script file by clicking **Run > Run**:

- If the script file contains a `main` function, the `main` function is executed, then execution stops. Only test cases and functions called by `main` will be executed, in the order in which they are specified in `main`.
- If the script does not contain a `main` function, the test cases are executed from top to bottom.

Example

The following template shows the structure of a script that contains a `main` function that passes data to a data-driven test case:

```
main ()
// 1. Declare a variable to hold current record
// 2. Store all data for test case in a list of records
// 3. Call the test case once for each record in the list
```

Using this structure, the following example shows how to create a script that defines data records and then calls the sample test case once for each record in the list:

```
type SEARCHINFO is record
  STRING sText      // Text to type in document window
  STRING sPos       // Starting position of search
  STRING sPattern   // String to look for
  BOOLEAN bCase     // Case-sensitive or not
  STRING sDirection // Direction of search
  STRING sExpected  // The expected match

main ()
  SEARCHINFO Data
  list of SEARCHINFO lsData = {...}
    {"Test Case", "<END>", "C", TRUE, "Up", "C"}
    {"Test Case", "<END>", "Ca", TRUE, "Up", "Ca"}
    // additional data records can be added here
  for each Data in lsData
    FindTest (Data)

testcase FindTest (SEARCHINFO Data)
  TextEditor.File.New.Pick ()
  DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
  TextEditor.Search.Find.Pick ()
```

```

Find.FindWhat.SetText (Data.sPattern)
Find.CaseSensitive.SetState (Data.bCase)
Find.Direction.Select (Data.sDirection)
Find.FindNext.Click ()
Find.Cancel.Click ()
DocumentWindow.Document.VerifySelText ({Data.sExpected})
TextEditor.File.Close.Pick ()
MessageBox.No.Click ()

```

When you click **Run > Run**, the main function is called and the `FindTest` test case will be executed once for every instance of `Data` in `IsData` (the list of `SEARCHINFO` records). In the script shown above, the test case will be run twice. Here is the results file that is produced:

```

Script findtest.t - Passed
Passed: 2 tests (100%)
Failed: 0 tests (0%)
Totals: 2 tests, 0 errors, 0 warnings

Testcase FindTest ({ "Test Case", "<END>", "C", TRUE, "Up", "C" }) - Passed
Testcase FindTest ({ "Test Case", "<END>", "Ca", TRUE, "Up", "Ca" }) - Passed

```



Note: With data-driven test cases, Silk Test Classic records the parameters that are passed in, in the results file.

In this sample data-driven test case, the test case data is stored in a list within the script itself. It is also possible to store the data externally and read records into a list using the `FileReadValue` function.

Using `do...except` to Handle an Exception

The `VerifyValue` method, like all 4Test verification methods, raises an exception if the actual value does not match the expected (baseline) value. When this happens, Silk Test Classic halts the execution of the test case and transfers control to the recovery system. The recovery system then returns the application to the base state.

However, suppose you don't want Silk Test Classic to transfer control to the recovery system, but instead want to trap the exception and handle it yourself. For example, you might want to log the error and continue executing the test case. To do this, you can use the 4Test `do...except` statement and related statements, which allow you to handle the exception yourself.

Select Data Source Dialog Box

The **Select Data Source** dialog box displays when you are creating a new data-driven file, which is a `.g.t` file, or opening an existing data-driven file that does not contain a data source connection string. This dialog box identifies the data source you will use to replace values in the data-driven test case. After you fill out this dialog box, Silk Test Classic loads the table and column information from your data source so it is available for you to use when you find and replace values in a test case. A data-driven script can use only one data source.

Select Data Source

Defaults to the previously specified DSN or to **Silk DDA Excel**. Select another data source by clicking the down arrow.

The **Select Data Source** dialog box allows you to choose either the **Silk DDA Excel** or the **Segue DDA Excel** data source. For new data-driven test cases, choose the **Silk DDA Excel** data source. Choose the **Segue DDA Excel** data source for backward compatibility. This allows existing `.g.t` files that reference **Segue DDA Excel** to continue to run.

Select Workbook/

Select Workbook displays when you have selected the default data source, **Segue DDA Excel**. **Connection String** displays when you choose a data source other than the

Connection String area	default. If you are working with Silk DDA Excel you must select a workbook. Click Browse to navigate to the workbook you will use.
OK	Saves the connection string so that it can be inserted into the data-driven script. The connection string is also saved to <code>partner.ini</code> so that this data source will be the new default selection for the list. If you are using the Silk DDA Excel data source, this button is only available after you select a workbook.
Customize	Displays when you select a data source other than the default. Click to display the Setup Data Driven Script DSN dialog box.

Setup Data Driven Script DSN Dialog Box

Use to create or modify the connection string for the DSN you select. Once you fill out this dialog, Silk Test Classic verifies that the DSN configuration is correct by connecting to the database, generating the 4Test code describing the DSN, and writing information into the data driven script.

Click **Customize** on the **Select Data Source** dialog box.

Source Provider	Select Microsoft OLE DB Provider for ODBC, since Silk Test Classic uses ODBC to connect to data sources.
------------------------	--

Microsoft OLE DB Provider for ODBC	Select Connect Using Data Source Name (DSN) .
---	--

Connection Information	
-------------------------------	--

DSN	
------------	--

	Select the data source that you selected on the Select Data Source dialog box.
--	---

The rest of the connection information displays information that the DSN requires before it can successfully make a connection. This information varies depending on the DSN you select. For more information about how to respond to the fields on this dialog, consult your database documentation or contact your database administrator.

More>

Click to display the connection string.

If you choose an MS Excel DSN and use the **Database** field to specify a workbook, then the Connection String will incorrectly contain `Database=<.xls file>`. To correct this problem, cut the `.xls` file path out of the **Database** field and paste it into the **Additional Connection String Parameters** field, preceded by `DBQ=`. For example, `DBQ=C:\TestExcel.xls`.

Specify Data Driven Script Dialog Box

You can use the **Specify Data Driven Script** dialog box to perform one or more of the following actions:

- To create a new data-driven script.
- To overwrite an existing data-driven script.
- To open an existing data-driven script and to append a test case to the script.

To open the **Specify Data Driven Script** dialog box, click **Tools > Data Drive Testcase** in the Silk Test Classic menu, and select a test case from the **Select Testcase** dialog box.



Note: The dialog box is available only when a script is open in the editor.

The dialog box includes the following elements:

Create a new file/ Overwrite an existing file	Click to create a new file or to overwrite the existing file that is specified in the Filename field. Silk Test Classic automatically adds the <code>.g.t</code> extension to the file name. You cannot overwrite existing files which are open in the editor.
Open an existing file	Click to open the existing file that is specified in the Filename field.
Filename	Type the name of the new file you want to create or the existing file you want to open or overwrite.
Browse	Click to navigate to the existing file you want to open or overwrite.
Save as format	You can select one of the following file formats: <ul style="list-style-type: none"> • ANSI • Unicode • UTF-8

Specify Data Driven Testcase Dialog Box

Use this dialog box to add a new data-driven test case or to overwrite existing test case(s).

Specify a data driven script file and, if prompted, a data source.

Add new Data Driven testcase	Click to create a new data driven test case. This new test case is appended to the end of the file and given the "DD_" prefix.
Overwrite existing Data Driven testcase	Click to overwrite an existing data driven test case.
Name	Defaults to <code>DD_<name of original testcase></code> when you are adding a new data driven test case. If you opt to overwrite an existing data-driven test case, this field turns into a list of data-driven test cases that already exist in the active file. Test case names can have a maximum of 127 characters. When you create a data-driven test case, Silk Test Classic truncates any test case name that is greater than 124 characters.

Specify Rows Dialog Box

Use to select rows to run the test case against. You use this dialog box to build queries that run against your data-driven test case.

Click **Specify Rows** on the **Run Testcase** dialog box or the **Testplan Detail** dialog box.

You cannot run using the sample record for individual tables. In order to run using the sample records for each table used by the test case, for example to do a quick test or if you are not connected to the database, click **Use Sample Data from Script** on the **Run Testcase** dialog box.

Table-by-table basis

When you select rows on a table-by-table basis, you use 'AND' and 'OR' actions. These may be used only to specify multiple conditions for a single table. You cannot include more than one table in a single 'where' clause. Conditions on separate tables must be specified independently, by selecting each table in the 'Table' list separately. The number of test cases that runs is:

of rows selected for Table 1 X the # of rows selected for Table 2 X the # of rows for Table 3

Table Lists the tables used by the test case to be run. For tables that are Microsoft Office Excel worksheets, the record names end in a dollar sign. Select a table in order to specify its where clause.

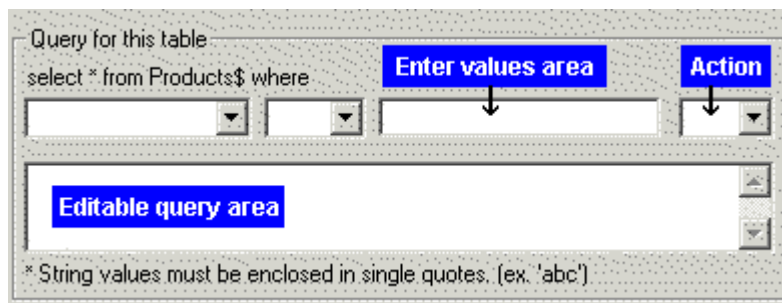
Run Runs the selected test case using the queries you have specified.

Debug Opens the file in which the test case resides, enters debugging mode and places a breakpoint at the first line of the test case. Debugging mode makes available all the debugging commands on the Breakpoint, Debug and View menus.

Query for this table area Guides you to enter the where clause for the selected table. If you leave this area blank and simply click **Run**, the default query selects all rows for the table.

The where clause of a query has the form:

```
(<Column> <relational operator> <value>) [and|or] (<Column> <relational operator> <value>)...
```



select * from <tablename> where the first list contains a list of columns from the table selected above.

where (the second list) lists the following relational operators that apply to all column types:

To indicate	Select
Equals	=
Not equal to	<>
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=
Like (typically used with STRING columns)	like

There are three additional relational operators that you can type into the Editable query area – but they do not appear in the where dropdown.

To Indicate	Type
Between two values (value1 <= value <= value2)	between <value1> and <value2>
In a set of discrete values	in (<value>, <value>, <value>)
Not in a set of discrete values	not in (<value>, <value>, <value>)

Enter values area

All string values must be enclosed in single quotation marks, for example, 'value'. For string columns, you can match a pattern by using like 'pattern'. The pattern can contain the single character (_) wildcard or the 0 or more characters (%) wildcard.

If a column allows NULL values, then you can use the following query to return only those rows for which the column contains a value:

```
select * from table_name where column_name <> NULL
```

Similarly, the following query would return only those rows for which the column is empty:

```
select * from table_name where column_name = NULL
```

Action Select **Done** to indicate you have finished entering where clauses or select **AND** or **OR** to indicate the relationship to another where clause. After you selected the appropriate action, the where clause that you have created appears in the bottom box on this dialog. You can add additional where clauses or click **Done**.

Editable query area

The where clauses that you create appear here after you click **DONE**, **AND**, or **OR**. You can add additional where clauses in this area by clicking and typing them in.

Sample queries

The following table shows several sample queries and their corresponding "where" clause.

To find	Use this WHERE clause
adults whose first names begin with 'B'	(FirstName like 'B%') and (Age >= 18)
bananas, cherries, apples, or any type of berry	(Fruit in ('banana', 'cherry', 'apple')) or (Fruit like '%berry')
people born in the 1960's with 3-letter last names	(YearOfBirth between 1960 and 1969) and (LastName like '____')

Go to Testcase Dialog Box

When a data-driven test script file, which is a file with the ending .g.t, is open in the editor, the **Go to Testcase** dialog box lists all the test cases that are available in the active script. Any listed data-driven test cases are prefixed with DD_.

To open the dialog box, click with the right mouse button into a .g.t file and select **Go to Testcase**.

Select the test case and then click **OK** to display the **4Test Editor** with the cursor at the beginning of the test case that you have selected.

Select Testcase Dialog Box

When creating a data-driven test case out of an existing test case, the **Select Testcase** dialog box lists the non-data-driven test cases in the active script.

To open the dialog box, click **Tools > Data Driven Testcase**. You can also right click in a .t file and then click **Data Driven Testcase**.

Select the test case that you want to data-drive and then click **OK** to display the **Specify Data Driven Script File** dialog box.

Characters Excluded from Recording and Replaying

The following characters are ignored by Silk Test during recording and replay:

Characters	Control
...	MenuItem
tab	MenuItem
&	All controls. The ampersand (&) is used as an accelerator and therefore not recorded.

Testing in Your Environment with the Open Agent

This section describes how you can test applications in your environment with the Open Agent.

Distributed Testing with the Open Agent

This section describes how you can run tests with the Open Agent on multiple machines.

Remote Testing with the Open Agent

You can install Silk Test Classic on a remote machine and test an application on this remote location from the Silk Test Classic that is installed on your local machine.



Note: If you want to test mobile applications on a mobile device that is connected to a remote machine or on an Emulator or Simulator on a remote machine, or web applications on Apple Safari or on a remote Microsoft Edge, you have to use a remote Silk Test information service instead of a remote Open Agent.

Testing with a Remote Open Agent

This functionality is supported only if you are using the Open Agent.

To replay tests against an application on a remote machine with Silk Test Classic, perform the following actions:

1. Create a test against the application on the local machine.
2. Install the Open Agent on the remote machine.
For additional information, refer to the [Silk Test Installation Guide](#).
3. Start the Open Agent on the remote machine.
4. Connect to the Open Agent on the remote machine.
 - Click **Options > Runtime** and specify the IP address of the remote machine in the **Agent name** field.
 - Add the `Connect` function to your test script.

For example, to connect to the Open Agent on the remote machine with the network name `agentmachine`, type the following into the test script:

```
[ ] HANDLE hOpenAgent
[ ] hOpenAgent = Connect("agentmachine", OPEN_AGENT)
```

To execute a command in the test script on the remote machine, add the handle to the command. For example, if the command in the local test script was `SomeAgentFunction`, change the command to the following:

```
[ ] hOpenAgent->SomeAgentFunction()
```

For additional information, see [Connect Function](#).

Connecting to a Remote Open Agent

This functionality is supported only if you are using the Open Agent.

You can only use TCP/IP to connect Silk Test Classic on the local machine with an Open Agent on a remote machine. Silk Test Classic on Microsoft Windows can use any TCP/IP software package that supports the Windows Sockets Interface Standard, Version 1.1, (WINSOCK), and supplies `WINSOCK.DLL`.

Networking Protocols Used by the Open Agent

The Open Agent uses exclusively the TCP/IP protocol.

Single Remote Applications

In a single-application test environment, if the application is remote, specify the agent name in the **Runtime Options** dialog box. This causes Silk Test Classic to automatically connect to that machine and to direct all agent commands to that machine. This contrasts with the multi-application case, in which you explicitly connect to the target machines and explicitly specify which machines are to receive which sections of code.

Multiple Remote Applications

When you enable networking by selecting the networking type in the **Runtime Options** dialog box on the host, do not set the **Agent Name** text box to an agent name if you have multiple remote agents. This field only accepts a single agent name and using it prevents you from handling all your client machines the same way.

If you specify one agent name from your set of agents, then you cannot issue a `Connect` call to that agent and thus do not receive the machine handle that the `Connect` function returns. Since you have to issue some `Connect` calls, be consistent and avoid writing exception code to handle a machine that is automatically connected.

For projects or scripts that use the Classic Agent, you can specify multiple agents from within your script file by adding the following command line to the agent:

```
agent -p portNumber
```

Configuring a Network of Computers

To configure a network of computers so that they can run Silk Test Classic and the Silk Test Classic agents, perform the following steps:

1. Install, or have already running, networking protocols supported by Silk Test Classic.
2. Install Silk Test Classic on the host machine and the agent software on all target machines.
3. Establish connectability between host and agents.
This may be automatic or may require some setup, depending on the circumstances.
4. Enable networking on any target machines.
Use the **Agent** window, as described in *Enabling Networking and Assigning the Classic Agent Name and Port*.
5. Enable networking on the host machine.
Use the **Runtime Options** dialog box. Details may vary, depending on your configuration.
6. Gather the information that your test scripts need when making explicit connections.
For example, you can edit the agent names into a list definition and have your test plan pass the list variable name as an argument for test cases controlled by that plan. The test cases then pass each agent name to a `Connect` or `SetUpMachine` function and that function makes the explicit host-to-agent connection.

Configuration details are specific to the different protocols and operating systems you are using. In general, set up your Agents and make all adjustments to the `partner.ini` file or environment variables before enabling networking on the host machine.

Enabling Networking on a Remote Host

Once the protocol has been picked for any PC agents and the port settings are consistent, you can enable networking on the host.

Do this by choosing **Options > Runtime** and setting the port number and/or agent name. You can skip this step if you do not have to change the default port number and you are not specifying an agent name for a single-remote-application configuration.

Configuring Open Agent Port Numbers

Typically, you do not have to configure port numbers manually. The information service handles port configuration automatically. Use the default port of the information service to connect to the agent. Then, the information service forwards communication to the port that the agent uses. However, if you have a port number conflict or an issue with a firewall, you must configure the port number for that machine or for the information service.

The default port of the information service is 22901. When you can use the default port, you can type `hostname` without the port number for ease of use. If you do specify a port number, ensure that it matches the value for the default port of the information service or one of the additional information service ports. Otherwise, communication will fail.

After changing the port number, restart the Open Agent, Silk Test Classic, Silk Test Recorder, and the application that you want to test.

Client/Server Testing Configurations

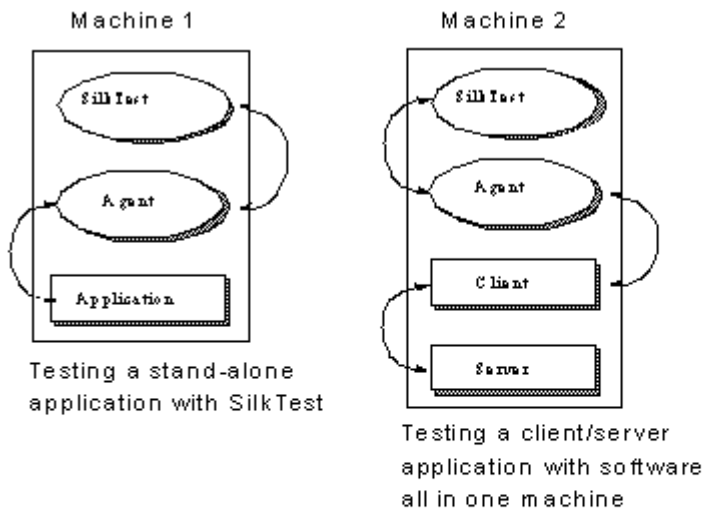
The processes that participate in a client/server testing scenario are logically associated with three different computers:

1. System A runs Silk Test Classic, which processes test scripts and sends application commands to the agent.
2. System B runs the client application and the agent, which submits the application commands to the client application.
3. System C runs the server software, which reacts to requests submitted by the client application.

The following sections describe different hardware/software configurations that can support Silk Test Classic testing.

Configuration 1

Machine 1 shows the software configuration you would have when testing a stand-alone application. Machine 2 shows Silk Test Classic and a client/server application with all of your software running on one machine. This configuration allows you to do all types of functional testing other than testing the behavior of the connection between a client and a remote server.



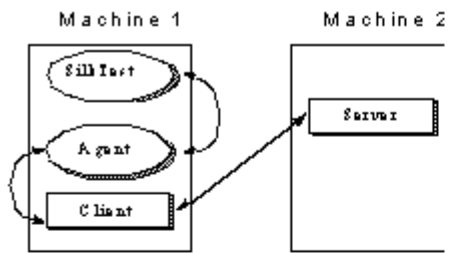
During your initial test development phase, you can reduce your hardware needs by making two (and possibly all) of these systems the same. If you write tests for an application running on the same system as Silk Test Classic, you can implement the tests without consideration of any of the issues of remote testing. You can then expand your testing program incrementally to take your testing into each new phase.

Configuration 2

A testing configuration in which the client application runs on the same machine as Silk Test Classic and the server application runs on a separate machine.



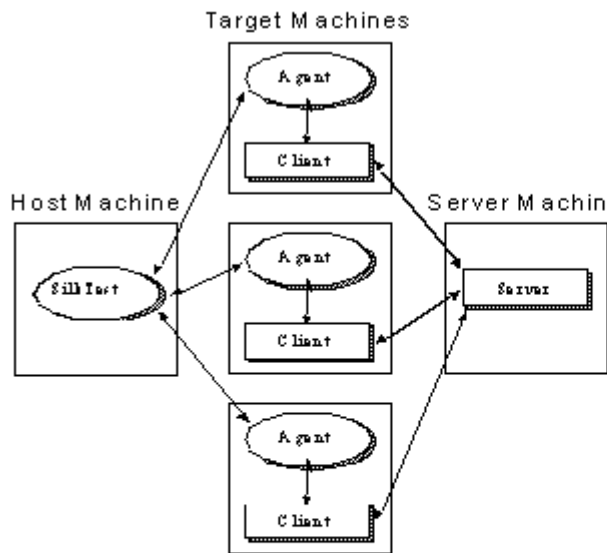
Note: In this configuration, as with Machine 2 in Configuration 1, there is no communication between Silk Test Classic and the server. This means that you must manage the work of starting and initializing the server database manually. For some kinds of testing this is appropriate.



This configuration lets you test the remote client-to-server connection and is appropriate for many stress tests. It allows you to do volume load testing from the point of view of the client application, but not the server.

Configuration 3

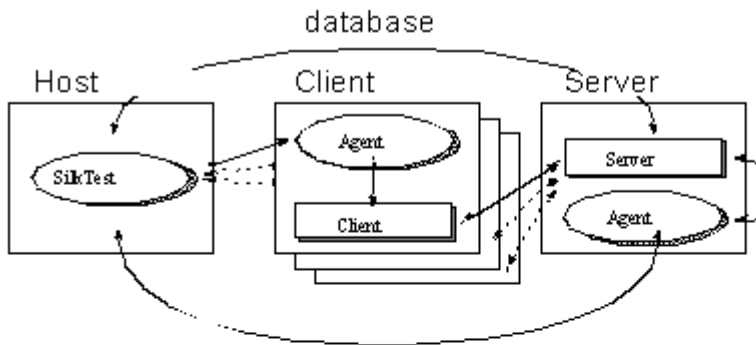
Multiple copies of the client application running on separate machines, with Silk Test Classic driving the client application by means of the agent process on each client machine, and the client application driving the server application. This is just the multi-client version of the previous configuration. You could run a fourth instance of the client application on the Silk Test Classic machine. The actual number of client machines used is your choice.



This configuration is appropriate for load testing and configuration testing if you have no need to automatically manipulate the server. You must have at least two clients to test concurrency and mutual-exclusion functionality.

Configuration 4

Once you are running Silk Test Classic, it makes sense to have your script initialize your server automatically. Configuration 4 uses the same hardware configuration as Configuration 3, but Silk Test Classic is also driving the server directly. This figure shows Silk Test Classic using an agent on the server machine to drive the server's GUI (the lower connecting arrow); this approach can be used to start the server's database and sometimes can be used to initialize it to a base state. The upper arrow shows Silk Test Classic using SQL commands to directly manipulate the server database; use this approach when using the agent is not sufficient. After starting the database with the agent, use SQL commands to initialize it to a base state. The SQL commands are submitted by means of Silk Test Classic's database functions, which do not require the services of the agent.



Configuration 4 is the most complete testing configuration. It requires the database tester. You can use it for all types of Silk Test Classic testing, including volume load testing of the server, peak load testing, and performance testing.

The special features that allow Silk Test Classic to provide rigorous testing for client/ server applications are the following:

- Automatic control of multiple applications.
- Multithreading for automatic control of concurrent applications.
- Reporting results by thread ID.
- Testing across networks using a variety of protocols.

The added value that the database tester provides for the client/server environment is direct database access from the test script.

Running Test Cases in Parallel

A concurrent, or multithreaded, script is one in which multiple statements can execute in parallel. Concurrency allows you to more effectively test distributed systems, by permitting multiple client applications to submit requests to a server simultaneously.

The 4Test language fully supports the development of concurrent scripts which enables a script to:

- Create and coordinate multiple concurrent threads.
- Protect access to variables, which are global to all threads.
- Synchronize threads with semaphores.
- Protect critical sections of code for atomic operations.
- Recover from errors in the event of script deadlock.

Concurrency

For Silk Test Classic, concurrent processing means that agents on a specified set of machines drive the associated applications simultaneously. To accomplish this, the host machine interleaves execution of the sets of code assigned to each machine. This means that when you are executing identical tests on several

machines, each machine can be in the process of executing the same operation. For example, select the **Edit.FindChange** menu item.

At the end of a set of concurrent operations, you will frequently want to synchronize the machines so that you know that all are ready and waiting before you submit the next operation. You can do this easily with 4Test.

There are several reasons for executing test cases concurrently:

- You want to save testing time by running your functional tests for all the different platforms at the same time and by logging the results centrally, on the host machine.
- You are testing cross-network operations.
- You need to place a multi-user load on the server.
- You are testing the application's handling of concurrent access to the same database record on the server.

To accomplish testing concurrent database accesses, you simply set all the machines to be ready to make the access and then you synchronize. When all the machines are ready, you execute the operation that commits the access operation—for example, clicking **OK**. Consider the following example:

```
// [A] Execute 6 operations on all machines concurrently
for each sMachine in lsMachine
    spawn
        SixOpsFunction (sMachine)
rendezvous // Synchronize

// [B] Do one operation on each machine
for each sMachine in lsMachine
    spawn
        [sMachine]MessageBox.OK.Click () // One operation
rendezvous // Synchronize
```

In code fragment [A], the six operations defined by the function `SixOpsFunction` are executed simultaneously on all machines in a previously defined list of agent names. After the parallel operation, the script waits for all the machines to complete; on completion, they will present a message box, unless the application fails. In code fragment [B], the message box is dismissed. By putting the message dismissal operation into its own parallel statement block instead of adding it to the `SixOpsFunction`, you are able to synchronize and all machines click at almost the same instant.

In order to specify that a set of machines should execute concurrently, you use a 4Test command that starts concurrent threads. In the fragments above, the `spawn` statement starts a thread for each machine.

Global Variables

Suppose the code for each machine is counting instances of some event. You want a single count for the whole test and so each machine adds its count to a global variable. When you are executing the code for all your machines in parallel, two instances of the statement `iGlobal = iGlobal + iCount` could be executing in parallel. Since the instructions that implement this statement would then be interleaved, you could get erroneous results. To prevent this problem, you can declare a variable shareable. When you do so, you can use the `access` statement to gain exclusive access to the shared variable for the duration of the block of code following the `access` statement. Make variables shareable whenever the potential for conflict exists.

Recovering Multiple Tests

There are three major categories of operations that an Agent executes on a target machine:

- Setup operations that bring the application to the state from which the next test will start.
- Testing operations that exercise a portion of the application and verify that it executed correctly.
- Cleanup operations that handle the normal completion of a test plus the case where the test failed and the application is left in an indeterminate state. In either case, the cleanup operations return the application to a known base state.

When there are multiple machines being tested and more than one application, the Agent on each machine must execute the correct operations to establish the appropriate state, regardless of the current state of the application.

Remote Recording

Once you establish a connection to a target machine, any action you initiate on the host machine, which is the machine running Silk Test Classic, is executed on the target machine.

With the Classic Agent, one Agent process can run locally on the host machine, but in a networked environment, the host machine can connect to any number of remote Agents simultaneously or sequentially. You can record and replay tests remotely using the Classic Agent. If you initiate a Record/Testcase command on the host machine, you record the interactions of the user manipulating the application under test on the target machine. In order to use the Record menu's remote recording operations, you must place the target machine's name into the **Runtime Options** dialog box. Choose **Options > Runtime**.

With the Open Agent, one Agent process can run locally on the host machine. In a networked environment, any number of Agents can replay tests on remote machines. However, you can record only on a local machine.

Threads and Concurrent Programming

Silk Test Classic can run test cases in parallel on more than one machine. To run test cases in parallel, you can use parallel threads within `main()` or in a function called by `main()`. If you attempt to run test cases in parallel on the same machine, you will generate a runtime error.

A more elegant alternative to parallel threads is to use a `multitestcase` function, which provides a robust multi-machine recovery system. For additional information on `multitestcase` code templates, see *Using the Client/Server Template* and *Using the Parallel Template*.

In the 4Test environment, a thread is a mechanism for interleaving the execution of blocks of client code assigned to different Agents so that one script can drive multiple client applications simultaneously. A thread is part of the script that starts it, not a separate script. Each thread has its own call stack and data stack. However, all the threads that a script spawns share access to the same global variables, function arguments, and data types. A file that one thread opens is accessible to any thread in that script.

While the creation of a thread carries no requirement that you use it to submit operations to a client application, the typical reason for creating a multithread script is so that each thread can drive test functions for one client, which allows multiple client application operations to execute in parallel.

When a script connects to a machine, any thread in that script is also connected to the machine. Therefore, you must direct the testing operations in a thread to a particular Agent machine. Threads interleave at the machine instruction level; therefore, no single 4Test statement is atomic with respect to a statement in another thread.

Driving Multiple Machines

When you want to run tests on multiple machines simultaneously, you connect to all the machines and then you direct specific test operations to particular machines. This enables you to drive different applications concurrently. For example, you can test the intercommunication capabilities of two different applications or you can drive both a client application and its server.

To do this, at the beginning of a test script you issue for each machine an explicit connection command. This can be either `Connect(agent_name)` or `SetMachine(agent_name)`. This connection lasts for the duration of the script unless you issue a `Disconnect(agent_name)` command. In the body of the script you can specify that a particular portion of code is to be executed on a particular machine. The `SetMachine(agent_name)` command specifies that the following statements are directed to that Agent. You can specify that just one statement is directed to a particular Agent by using the bracket form of the machine handle operator. For example `["Client_A"]SYS_SetDir ("c:\mydir")`.

Since 4Test allows you to pass variables to these functions, you can write a block of code that sends the same operations to a particular set of target machines and you can pass the `SetMachine` function in that block of code a variable initialized from a list that specifies the machines in that set. Thus, specifying which machines receive which operations is very simple.

Protecting Access to Global Variables

When a new thread is spawned, 4Test creates a new copy of all local variables and function arguments for it to use. However, all threads have equal access to global variables. To avoid a situation in which multiple threads modify a variable simultaneously, you must declare the variable as shareable. A shareable variable is available to only one thread at a time.

Instances where threads modify variables simultaneously generate unpredictable results. Errors of this kind are difficult to detect. Make variables shareable wherever the potential for conflict exists.

A declaration for a shareable variable has the following form:

```
[scope] share data-type name [= expr] {, name [= expr]}
```

- *scope* can be either public or private. If omitted, the default is public.
- *data-type* is a standard or user-defined data type.
- *name* is the identifier that refers to the shareable variable.
- *expr* is an expression that evaluates to the initial value you want to give the variable. The value must have the same type you gave the variable. If you try to use a variable before its value is set, 4Test raises an exception.

At any point in the execution of a script, a shared variable can only be accessed from within the block of code that has explicitly been granted access to it. You request access to shareable variables by using the access statement.

An access statement has the following form:

```
access name1, name2, ...  
    statement
```

where *name1*, *name2*, ... is a list of identifiers of optional length, each of which refers to a shareable variable and *statement* is the statement to be executed when access to the variables can be granted.

If no other thread currently has access to any of the shareable variables listed, 4Test executes the specified statement. Otherwise, 4Test blocks the thread where the `access` statement occurs until access can be granted to all the shareable variables listed. At that point, 4Test blocks competing threads and executes the blocked thread.

Example

```
share INTEGER iTestNum = 0  
public share STRING asWeekDay [7]  
share ANYTYPE aWhoKnows  
  
void IncrementTestNum ()  
    access iTestNum  
    iTestNum = iTestNum + 1
```

Synchronizing Threads with Semaphores

Use semaphores to mutually exclude competing threads or control access to a resource. A semaphore is a built-in 4Test data type that can only be assigned a value once. The value must be an integer greater than zero. Once it is set, your code can get the semaphore's value, but cannot set it.

Example

The following code example shows legal and illegal manipulations of a variable of type *SEMAPHORE*:

```
SEMAPHORE semA = 10 // Legal
semA = 20 // Illegal -
existing semaphore // cannot be
reinitialized
if (semA == [SEMAPHORE]2)... // Legal - note the
typecast
Print ("SemA has {semA} resources left.") // Legal
SEMAPHORE semB = 0 // Illegal - must be
greater than 0
```

To compare an integer to a semaphore variable, you must typecast from integer to semaphore using `[SEMAPHORE]`.



Note: You cannot cast a semaphore to an integer.

To use a semaphore, you first declare and initialize a variable of type *SEMAPHORE*. Thereafter, 4Test controls the value of the semaphore variable. You can acquire the semaphore if it has a value greater than zero. When you have completed your semaphore-protected work, you release the semaphore. The `Acquire` function decrements the value of the semaphore by one and the `Release` function increments it by one. Thus, if you initialize the semaphore to 5, five threads can simultaneously execute semaphore-protected operations while a sixth thread has to wait until one of the five invokes the `Release` function for that semaphore.

The `Acquire` function either blocks the calling thread because the specified semaphore is zero, or "acquires" the semaphore by decrementing its value by one. `Release` checks for any threads blocked by the specified semaphore and unblocks the first blocked thread in the list. If no thread is blocked, `Release` "releases" the semaphore by incrementing its value by one so that the next invocation of `Acquire` succeeds, which means it does not block.

A call to `Acquire` has the following form:

```
void Acquire(SEMAPHORE semA)
```

Where `semA` is the semaphore variable to acquire.

A call to `Release` has the following form:

```
void Release(SEMAPHORE semA)
```

Where `semA` is the semaphore variable to release.

If more than one thread was suspended by a call to `Acquire`, the threads are released in the order in which they were suspended.

A semaphore that is assigned an initial value of 1 is called a binary semaphore, because it can only take on the values 0 or 1. A semaphore that is assigned an initial value of greater than one is called a counting semaphore because it is used to count a number of protected resources.

Example: Application only supports three simultaneous users

Suppose you are running a distributed test on eight machines using eight 4Test threads. Assume that the application you are testing accesses a database, but can support only three simultaneous users. The following code uses a semaphore to handle this situation:

```
SEMAPHORE DBUsers = 3
...
```

```
Acquire (DBUsers)
    access database
Release (DBUsers)
```

The declaration of the semaphore is global; each thread contains the code to acquire and release the semaphore. The initial value of three ensures that no more than three threads will ever be executing the database access code simultaneously.

Testing In Parallel but Not Synchronously

This topic illustrates a method for running test functions in parallel on multiple clients, but with different tests running on each client. This provides a realistic multi-user load as opposed to a load in which all clients perform the same operations at roughly the same time.

Example

This example suggests a method by which each client, operating in a separate thread, executes a test that is assigned by a random number. The `RandSeed` function is called first so that the random number sequence is the same for each iteration of this multi-user test scenario. This enables you to subsequently repeat the test with the same conditions.

The example reads a list of client machines from a file, `clients.txt`, and receives the test count as in input argument. These external variables make the example scalable as to the number of machines being tested and the number of tests to be run on each. The number of different testcases is twelve in this example, but could be changed by modifying the `SelectTest` function and adding further test functions. For each machine in the client machine list, the example spawns a thread in which the specified client executes a randomly selected test, repeating for the specified number of tests.



Note: You can execute this test as it is written because it sets its own application states. However, when you use multi-application support, this is automatic. And if you want to use this approach to drive different applications or to initialize a server before starting the testing, you must add multi-application support.

```
testcase ParallelRandomLoadTest (INTEGER iTestCount)
    LIST OF STRING lsClients
    RandSeed (3)

    // list of client names
    ListRead (lsClients, "clients.txt")

    STRING sClientName

    for each sClientName in lsClients
        spawn
            // Connect to client, which becomes current machine
            Connect (sClientName)
            SetAppState ("MyAppState") // Initialize
application
            TestClient (iTestCount)
            Disconnect (sClientName)
        rendezvous

    TestClient (INTEGER iTestCount)
        for i = 1 to iTestCount
            SelectTest ()

    SelectTest ()
```

```

INTEGER i = RandInt (1, 12)

// This syntax invokes Test1 to Test12, based on i
@("Test{i}") ()

// Define the actual test functions
Test1 ()
    // Do the test . . .

Test2 ()
    // Do the test . . .
. . .
Test12 ()
    // Do the test . . .

```

Statement Types

This section describes the statement types that are available for managing distributed tests.

Parallel Processing Statements

You create and manage multiple threads using combinations of the 4Test statements `parallel`, `spawn`, `rendezvous`, and `critical`.

In 4Test, all running threads, which are those not blocked, have the same priority with respect to one another. 4Test executes one instruction for a thread, then passes control to the next thread. The first thread called is the first run, and so on.

All threads run to completion unless they are deadlocked. 4Test detects script deadlock and raises an exception.



Note: The 4Test exit statement terminates all threads immediately when it is executed by one thread.

Using Parallel Statements

A `parallel` statement spawns a statement for each machine specified and blocks the calling thread until the threads it spawns have all completed. It condenses the actions of `spawn` and `rendezvous` and can make code more readable.

The `parallel` statement executes a single statement for each thread. Thus if you want to run complete tests in parallel threads, use the invocation of a test function, which may execute many statements, with the `parallel` statement, or use a block of statements with `spawn` and `rendezvous`.

To use the `parallel` statement, you must specify the machines for which threads are to be started. You can follow the `parallel` keyword with a list of statements, each of which specifies a different Agent name. For example:

```

parallel
    DoSomething ("Client1")
    DoSomething ("Client2")

```

The `DoSomething` function then typically issues a `SetMachine(sMachine)` call to direct its machine operations to the proper Agent.

Using a Spawn Statement

A `spawn` statement begins execution of the specified statement or block of statements in a new thread. Since the purpose of `spawn` is to initiate concurrent test operations on multiple machines, the structure of a block of spawned code is typically:

- A `SetMachine` command, which directs subsequent machine operations to the specified agent.

- A set of machine operations to drive the application.
- A verification of the results of the machine operations.

You can use `spawn` to start a single thread for one machine, and then use successive `spawn` statements to start threads for other machines being tested. Silk Test Classic scans for all `spawn` statements preceding a rendezvous statement and starts all the threads at the same time. However, the typical use of `spawn` is in a loop, like the following:

```
for each sMachine in lsMachine
  spawn          // start thread for each sMachine
    SetMachine (sMachine)
    DoSomething ()
  rendezvous
```

The preceding example achieves the same result when written as follows:

```
for each sMachine in lsMachine
  spawn
    [sMachine]DoSomething ()
  rendezvous
```

To use a `spawn` statement in tests that use TrueLog, use the `OPT_PAUSE_TRUELOG` option to disable TrueLog. Otherwise, issuing a `spawn` statement when TrueLog is enabled causes Silk Test Classic to hang or crash.

Using Templates

This section describes how you can use templates for distributed testing.

Using the Parallel Template

This template is stored as `parallel.t` in the `Examples` subdirectory of the Silk Test Classic installation directory. The code tests a single application that runs on an externally defined set of machines.

This multi-test-case template accepts a list of machine names. The application whose main window is `MyMainWin` is invoked on each machine. The same operations are then performed on each machine in parallel. If any test case fails, the multi-test-case will be marked as having failed; however, a failed test case within a thread does not abort the thread.

You can use this template by doing three edits:

- Include the file that contains your window declarations.
- Substitute the `MainWin` name of your application, which is defined in your `MainWin` window declaration, with the `Mainwin` name of the template, `MyMainWin`.
- Insert the calls to one or more tests, or to the main function, where indicated.

Use `myframe.inc`.

```
use "myframe.inc"
multitestcase MyParallelTest (LIST of STRING lsMachines)

  STRING sMachine

  // Connect to all machines in parallel:
  for each sMachine in lsMachines
    spawn
      SetUpMachine (sMachine, MyMainWin)
    rendezvous

  // Set app state of each machine, invoking if necessary:
  SetMultiAppStates()

  // Run testcases in parallel
  for each sMachine in lsMachines
    spawn
```



```
SetMachine (sMachine)
// Call testcase(s) or call main()
rendezvous
```

Client/Server Template

This template is stored as `multi_cs.t` in the `Examples` subdirectory of the Silk Test Classic installation directory. This test case invokes the server application and any number of client applications, based on the list of machines passed to it, and runs the same function on all clients concurrently, after which the server will perform end-of-session processing.

You can use this template by doing the following edits:

- Include the files that contain your window declarations for both the client application and the server application.
- Substitute the `MainWin` name of your server application, which is defined in your `MainWin` window declaration, with the `MainWin` name of the template, `MyServerApp`.
- Substitute the `MainWin` name of your client application, which is defined in your `MainWin` window declaration, with the `Mainwin` name of the template, `MyClientApp`.
- Replace the call to `PerformClientActivity` with a function that you have written to perform client operations and tests.
- Replace the call to `DoServerAdministration` with a function that you have written to perform server administrative processing and/or cleanup.

```
use "myframe.inc"
multitestcase MyClientServerTest (STRING sServer, LIST of STRING lsClients)
    STRING sClient

    // Connect to server machine:
    SetUpMachine (sServer, MyServerApp)

    // Connect to all client machines in parallel:
    for each sClient in lsClients
        spawn
            SetUpMachine (sClient, MyClientApp)
    rendezvous

    // Set app state of each machine, invoking if necessary:
    SetMultiAppStates()

    // Run functions in parallel on each client:
    for each sClient in lsClients
        spawn
            // Make client do some work:
            [sClient] PerformClientActivity()
        rendezvous

    // Perform end-of-session processing on server application:
    [sServer] DoServerAdministration()
```

Testing Multiple Machines

This section describes strategies for testing multiple machines.

Running Tests on One Remote Target

Use one of the following methods to specify that you want a script, suite, or test plan to run on a remote target instead of the host:

- Enter the name of the target Agent in the **Runtime Options** dialog box of the host. You also need to select a network protocol in the dialog box. If you have been testing a script by running Silk Test Classic

and the Agent on the same system, you can then test the script on a remote system without editing your script by using this method.

- Specify the target Agent's name by enclosing it within brackets before the script or suite name. For example `[Ohio]myscript.t`.
- You can select `(none)` in the **Runtime Options** dialog box of the host and then specify the name of the target Agent in a call to the `Connect` function in your script. For example, to connect to a machine named Ontario:

```
testcase MyTestcase ()
  Connect ("Ontario")
  // Call first testcase
  DoTest1 ()
  // Call second testcase
  DoTest2 ()
  Disconnect ("Ontario")
```

When you are driving only one remote target, there is no need to specify the current machine; all test case code is automatically directed to the only connected machine.

When you use the multi-application support functions, you do not have to make explicit calls to `Connect`; the support functions issue these calls for you.

Running Tests Serially on Multiple Targets

To run your scripts or suites serially on multiple target machines, specify the name of the target Agent within the suite file. For example, the following code runs a suite of three scripts serially on two target machines named Ohio and Montana:

```
[Ohio] script1.t
[Ohio] script2.t
[Ohio] script3.t
[Montana] script1.t
[Montana] script2.t
[Montana] script3.t
```

Any spaces between the name of the target Agent and the script name are not significant.

Alternatively, to run test cases serially on multiple target machines, switch among the target machines from within the script, by using the `Connect` and `Disconnect` functions of 4Test. For example, the following script contains a function named `DoSomeTesting` that is called once for each machine in a list of target machines, with the name of the target Agent as an argument:

```
testcase TestSerially ()
  STRING sMachine
  // Define list of agent names
  LIST OF STRING lsMachines = {...}
  "Ohio"
  "Montana"

  // Invoke test function for each name in list
  for each sMachine in lsMachines
    DoSomeTesting (sMachine)

  // Define the test function
  DoSomeTesting (STRING sMachine)
    Connect (sMachine)
    Print ("Target machine: {sMachine}")
    // do some testing...
    Disconnect (sMachine)
```

You will rarely need to run one test serially on multiple machines. Consider this example a step on the way to understanding parallel testing.

Specifying the Target Machine Driven By a Thread

While the typical purpose for a thread is to direct test operations to a particular test machine, you have total flexibility as to which machine is being driven by a particular thread at any point in time. For example, in the code below, the `spawn` statement starts a thread for each machine in a predefined list of test machines. The `SetMachine` command directs the code in that thread to the Agent on the specified machine. But the `["server"]` machine handle operator directs the code in the `doThis` function to the machine named `server`. The code following the `doThis` invocation continues to be sent to the `sMachine` specified in the `SetMachine` command.

```
for each smachine in lsMachine
  spawn // start thread for each sMachine
  SetMachine (sMachine)
  // ... code executed on sMachine
  ["server"]doThis() // code executed on "server"
  // ...continue with code for sMachine
rendezvous
```

While the machine handle operator takes only a machine handle, `4Test` implicitly casts the string form of the Agent machine's name as a machine handle and so in the preceding example the machine name is effectively the same as a machine handle.

Specifying the Target Machine For a Single Command

To specify the target machine for a single command, use the machine handle operator on the command. For example, to execute the `SYS_SetDir` function on the target machine specified by the `sMachine1` variable, type `sMachine1->SYS_SetDir (sDir)`.

To allow you to conveniently perform system related functions (`SYS_`) on the host, you can preface the function call with the machine handle operator, specifying the globally defined constant `hHost` as the argument to the operator. For example, to set the working directory on the host machine to `c:\mydir`, type `hHost->SYS_SetDir ("c:\mydir")`.

You can use this syntax with a method call, for example `sMachine->TextEditor.Search.Find.Pick`, but when invoking a method, this form of the machine handle must be the first token in the statement.

If you need to use this kind of statement, use the alternative form of the machine handle operator described below.

You can use the `SetMachine` function to change target machines for an entire block of code.

The `hHost` constant cannot be used in simple handle compares like `hMyMachineHandle== hHost`. This will never be `TRUE`. A better method is to use `GetMachineName(hHost)` and compare names. If `hHost` is used as an argument, it will refer to the "(local)" host not the target host.

Example

The following example shows valid and invalid syntax:

```
// Valid machine handle operator use
for each sMachine in lsMachine
  sMachine-> TextEditor.Search.Find.Pick

// Invalid machine handle operator use with method
if (sMachine->ProjX.DuplicateAlert.Exists())
  Print ("Duplicate warning on {sMachine} recipient.")
```

If you need to use this kind of statement, use the alternative form of the machine handle operator described below.

You can use the `SetMachine` function to change target machines for an entire block of code.

The `hHost` constant cannot be used in simple handle compares, like `hMyMachineHandle == hHost`. This will never be `TRUE`. A better method is to use `GetMachineName(hHost)` and compare names. If `hHost` is used as an argument, it will refer to the local host, not the target host.

Reporting Distributed Results

You can view test results in each of several formats, depending on the kind of information you need from the report. Each format sorts the results data differently, as follows:

- Elapsed time** Sorts results for all threads and all machines in event order. This enables you to see the complete set of results for a time period and may give you a sense of the load on the server during that time period or may indicate a performance problem.
- Machine** Sorts results for all threads running on one machine and presents the results in time-sorted order for that machine before reporting on the next machine.
- Thread** Sorts results for all tests run under one thread and presents the results in time-sorted order for that thread before reporting on the next thread.

Alternative Machine Handle Operator

An alternative syntax for the machine handle operator is the bracket form, like the following example shows.

```
[hMachine] Any4TestFunctionCall ()
```

Example

To execute the `SYS_SetDir` function on the target machine specified by the string `sMachineA`, you do this:

```
[sMachineA] SYS_SetDir (sDir)
```

The correct form of the invalid syntax shown above is:

```
// Invalid machine handle operator use
if ([sMachine]ProjX.DuplicateAlert.Exists())
    Print ("Duplicate warning on {sMachine} recipient.")
```

To execute the `SYS_SetDir` function on the host machine, you can do the following:

```
[hHost] SYS_SetDir (sDir)
```

You can also use this form of the machine handle operator with a function that is not being used to return a value or with a method.

Example

```
for each sMachine in lsMachine
    [sMachine] FormatTest7 ()
```

Example

```
for each sMachine in lsMachine
    [sMachine] TextEditor.Search.Find.Pick
```

Testing Clients Concurrently

In concurrent testing, Silk Test Classic executes one function on two or more clients at the same time. This topic demonstrates one way to perform the same tests concurrently on multiple clients.

For example, suppose you want to initiate two concurrent database transactions on the same record, and then test how well the server performs. To accomplish this, you need to change the script presented in *Testing Clients Plus Server Serially* to look like this:

```
testcase TestConcurrently ()
  Connect ("server")
  Connect ("client1")
  Connect ("client2")
  DoSomeSetup ("server") // initialize server first
  Disconnect ("server") // testcase is thru with server

  spawn // start thread for client1
    UpdateDatabase ("client1")
  spawn // start thread for client2
    UpdateDatabase ("client2")

  rendezvous // synchronize
  Disconnect ("client1")
  Disconnect ("client2")

  DoSomeSetup (STRING sMachine) // define server setup
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to do server setup goes here
    TimerStop (hTimer)
    Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
    TimerDestroy (hTimer)

  UpdateDatabase (STRING sMachine) // define update test
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to update database goes here
    TimerStop (hTimer)
    Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
    TimerDestroy (hTimer)
```

An alternative but equivalent approach is to use the parallel statement in place of the spawn and rendezvous:

```
testcase TestConcurrently2 ()
  Connect ("server")
  Connect ("client1")
  Connect ("client2")

  DoSomeSetup ("server")
  Disconnect ("server")

  parallel // automatic synchronization
    UpdateDatabase ("client1") // thread for client1
    UpdateDatabase ("client2") // thread for client2

  Disconnect ("client1")
  Disconnect ("client2")

  DoSomeSetup (STRING sMachine)
    HTIMER hTimer
    hTimer = TimerCreate ()
    TimerStart (hTimer)
    SetMachine (sMachine)
    // code to do server setup goes here
```

```

TimerStop (hTimer)
Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
TimerDestroy (hTimer)

UpdateDatabase (STRING sMachine)
HTIMER hTimer
hTimer = TimerCreate ()
TimerStart (hTimer)
SetMachine (sMachine)
// code to update database goes here
TimerStop (hTimer)
Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
TimerDestroy (hTimer)

```

If you use variables to specify different database records for each client's database transactions, you can use the above techniques to guarantee parallel execution without concurrent database accesses.

Testing Clients Plus Server Serially

In a client/server application, the server and its clients typically run on different target machines. This topic explains how to build tests that test the server and its clients in a serial fashion. In this scenario, the `SetMachine` function switches among the target machines on which the server and its clients are running. The following script fragment tests a client/server database application in the following steps:

1. Connect to three target machines, which are server, client1, and client2.
2. Call the `DoSomeSetup` function, which calls `SetMachine` to make "server" the current target machine, and then perform some setup.
3. Call the `UpdateDatabase` function once for each client machine. The function sets the target machine to the specified client, then does a database update. It creates a timer to time the operation on this client.
4. Disconnect from all target machines.

Example

This example shows how you direct sets of test case statements to particular machines. If you were doing functional testing of one application, you might want to drive the server first and then the application. However, this example is not realistic because it does not show the support necessary to bring the different machines to their different application states and to recover from a failure on any machine.

```

testcase TestClient_Server ()
  Connect ("server")
  Connect ("client1")
  Connect ("client2")
  DoSomeSetup ("server")
  UpdateDatabase ("client1")
  UpdateDatabase ("client2")
  DisconnectAll ()

DoSomeSetup (STRING sMachine)
  HTIMER hTimer
  hTimer = TimerCreate ()
  TimerStart (hTimer)
  SetMachine (sMachine)
  // code to do server setup goes here
  TimerStop (hTimer)
  Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
  TimerDestroy (hTimer)

UpdateDatabase (STRING sMachine)
  HTIMER hTimer
  hTimer = TimerCreate ()

```

```
TimerStart (hTimer)
SetMachine (sMachine)
// code to update database goes here
TimerStop (hTimer)
Print ("Time on {sMachine} is: {TimerStr (hTimer)}")
TimerDestroy (hTimer)
```

Testing Databases

You may be testing a distributed application that accesses a database or you may be directly testing database software. In either of these cases, you might want to manipulate the database directly from Silk Test Classic for several purposes:

- To exercise certain database functions that are present in a GUI that runs directly on the server machine and is not a client application. For example, administrative functions used for setting up the database.
- To set the server database to a known state.
- To verify an application's database results without using the application.
- To read information from the database to use as input to a test case.

Silk Test Classic can drive a server application's GUI by means of the Silk Test Classic Agent exactly as it drives a client application. In addition, the database tester provides direct access, using SQL, from a test script to any database supported by ODBC drivers. These database functions enable you to read and write database records without using the client application. Thus, you can verify client test results without assuming the ability of the client to do that verification.

In addition to using the SQL functions in your tests, you can also use these functions to help manage your testing process as follows:

- Maintain a bug database, updating it with the results of your testing.
- Manage your test data in a database instead of in a text file.

The database functions, among other things, allow you to connect to a database, submit an SQL statement, read data from the selected record(s) if the SQL statement was SELECT, and subsequently disconnect from the database. About a dozen of these functions allow you to access your database's catalog tables.

The functions that support these operations begin with the letters "DB_".

Testing Multiple Applications

This section describes testing multiple applications.

Overview of Multi-Application Testing

Silk Test Classic can easily drive multiple different applications simultaneously. Thus you can bring a server's database to a known state at the same time you are bringing multiple instances of the client application to their base state window. Likewise, you can drive a server database with several different client applications at the same time.

The essential difference between single-application and multi-application testing is clearly the difference between "one" and "many." When the following entities in a test case are greater than one, they need special consideration and support functions found in Silk Test Classic:

- Agent names.
- Application main window names.
- Sets of application states associated with each main window name.

Multi-machine testing requires that you map both the name of an application and all application states for that application to the machine on which it will be tested. This makes it possible for you to direct test

operations to the right machines, and it enables Silk Test Classic to automatically set the machines to the proper application state before a test is run, and to clean up after a test has failed.

Test Case Structure in a Multi-Application Environment

This topic describes Silk Test Classic components that enable concurrent testing of more than one application. For example, there are functions that make it possible to drive both the client application and the client's server from Silk Test Classic, to set each to its base state, and to recover each if it fails. Compare with the test case structure of a single-application environment.

The multi-application environment uses the same `defaults.inc` file as does the single-application environment. However, when you define a function as a `multitestcase`, 4Test uses functions defined in the `cs.inc` file to invoke functions in `defaults.inc`. Thus, it can pass the appropriate application states or base states to these functions, depending on the requirements of a particular test machine.

Instead of preceding the test case function declaration with the keyword `testcase`, you must use the keyword `multitestcase` to give your test case the multi-application recovery system.

`cs.inc` is an automatically included file that contains functions used only in the multi-application environment. For additional information about this file and the functions that it contains, see *cs.inc*. You may need to include other files also.

Invoking a Test Case in a Multi-Application Environment

The keyword for a test case declaration is different when you are performing distributed testing. In the single-application environment, you invoke a test case with no arguments or you may specify an application state function. However, in a multi-application environment, instead of preceding the test case function declaration with the keyword `testcase`, you must use the keyword `multitestcase` to give your test case the multi-application recovery system.

Declaring a function as a `multitestcase` gives that function the ability to invoke functions declared with the keyword `testcase`. A `multitestcase` thus can be viewed as a wrapper for stand-alone test cases; it provides a means of assigning tests to particular machines and lets you invoke previously written test cases from the multi-test case file by simply adding a use statement to the file to include the test case definitions.

When you are using multi-application environment support, you can pass the test case the names of the machines to be tested during that execution of the test case, but not the application state function. In a multi-application environment, one test case can use multiple application states; you specify these in the required code at the beginning of the test case.

Test Case Structure in a Single-Application Environment

The code that implements a test case for a single application is similar to that of a test case for applications on multiple separate machines in a client/server environment.

This topic summarizes the structure of the single-application version and some Silk Test Classic components used to implement it. You can compare the structure with the support code needed for running multiple applications.

The include file `defaults.inc` implements the recovery system for a single application test. For information about the `DefaultBaseState` function and the functions that are contained within `defaults.inc`, see *defaults.inc*.

Your test case needs certain definitions that other test cases in your testing program will also need. These include:

- Window declarations
- Application states
- Utility functions

Placing these general purpose definitions in an include file, or several smaller files, saves repetitive coding. When you use Silk Test Classic to record window declarations and application states, Silk Test Classic names the generated file `frame.inc`.

Window Declarations for Multi-Application Testing

In the client/server environment, unlike the stand-alone environment, you can test two or more different applications at the same time. For example, you could run the functional tests for application "A" on multiple machines at the same time that you are running the functional tests for application "B" on the same machines. The include files that you must generate may therefore have to take into consideration different platforms and/or different applications.

When you are driving two or more applications from Silk Test Classic, you need separate window declarations for each different application. You must be certain that your main window declaration for each separate application is unique. If the same application is running on different platforms concurrently, you may need to use GUI specifiers to specialize the window declarations. 4Test will identify a window declaration statement, that is preceded by a GUI specifier, as being true only on the specified GUI.

In addition, you may find that the operations needed to establish a particular application state are slightly different between platforms. In this case, you just record application states for each platform and give them names that identify the state and the GUI for your convenience.

Recording window declarations on a client machine that is not the host machine, requires that you operate both Silk Test Classic on the host machine and the application on its machine at the same time. You record window declarations and application states in much the same way for a remote machine as for an application running in the Silk Test Classic host machine. The primary difference is that you start the recording operation by selecting Test Frame in Silk Test Classic on the host system and you do the actual recording of application operations on the remote system.

If you have two or more applications being tested in parallel, you need to have two or more sets of window declarations. You must have window declarations, and application states, if needed, for each different application. When recording window declarations and application states on a remote machine, you will find it convenient to have the machine physically near to your host system.

Remote Recording

This functionality is supported only if you are using the Classic Agent.

Concurrency Test Example Code

The concurrency test example is designed to allow any number of test machines to attempt to access a server database at the same time. This tests for problems with concurrency, such as deadlock or out-of-sequence writes.

This example uses only one application. However, it is coded in the style required by the multi-application environment because you will probably want to use an Agent to start and initialize the server during this type of test. There is no requirement in the client/server environment that you use the single-application style of test case just because you are driving only one application. For consistency of coding style, you will probably find it convenient to always use the multi-application files and functions.

For detailed information on the code example, see *Concurrency Test Explained*.

```
const ACCEPT_TIMEOUT = 15
multitestcase MyTest (LIST OF STRING lsMachine)
    STRING sMachine
    INTEGER iSucceed
    STRING sError

    for each sMachine in lsMachine
        SetUpMachine (sMachine, Personnel)
        SetMultiAppStates ( )

    /*** HAVE EACH MACHINE EDIT THE SAME EMPLOYEE ***/
```

```

for each sMachine in lsMachine
  spawn

  /** SET THE CURRENT MACHINE FOR THIS THREAD ***/
  SetMachine (sMachine)

  /** EDIT THE EMPLOYEE RECORD "John Doe" ***/
  Personnel.EmployeeList.Select ("John Doe")
  Personnel.Employee.Edit.Pick ()

  /** CHANGE THE SALARY TO A RANDOM NUMBER BETWEEN
  50000 AND 70000 ***/
  Employee.Salary.SetText ([STRING] RandInt (50000, 70000))
rendezvous

/** ATTEMPT TO HAVE EACH MACHINE SAVE THE EMPLOYEE RECORD ***/
for each sMachine in lsMachine
  spawn

  /** SET THE CURRENT MACHINE FOR THIS THREAD ***/
  SetMachine (sMachine)

  /** SELECT THE OK BUTTON ***/
  Employee.OK.Click ()

  /** CHECK IF THERE IS A MESSAGE BOX ***/
  if (MessageBox.Exists (ACCEPT_TIMEOUT))
    SetMachineData (NULL, "sMessage",
    MessageBox.Message.GetText ())
    MessageBox.OK.Click ()
    Employee.Cancel.Click ()
  else if (Employee.Exists ())
    AppError ("Employee dialog not
    dismissed after {ACCEPT_TIMEOUT} seconds")
rendezvous

/** VERIFY THE OF NUMBER OF MACHINES WHICH SUCCEEDED ***/
iSucceed = 0
for each sMachine in lsMachine
  sError = GetMachineData (sMachine, "sMessage")
  if (sMessage == NULL)
    iSucceed += 1
  else
    Print ("Machine {sMachine} got message '{sMessage}'")

Verify (iSucceed, 1, "number of machines that succeeded")

```

Concurrency Test Explained

Before you record and/or code your concurrency test, you record window declarations that describe the elements of the application's GUI. These are placed in a file named `frame.inc`, which is automatically included with your test case when you compile. Use Silk Test Classic to generate this file because Silk Test Classic does most of the work.

The following code sample gives just those window declarations that are used in the *Concurrency Test Example*:

```

window MainWin Personnel
  tag "Personnel"
  PopupList EmployeeList
  Menu Employee
    tag "Employee"
  MenuItem Edit
    tag "Edit"

```

```
// ...
window DialogBox Employee
  tag "Employee"
  parent Personnel
  TextField Salary
    tag "Salary"
  PushButton OK
    tag "OK"
// ...
```

The following explanation of the *Concurrency Test Example* gives the testing paradigm for a simple concurrency test and provides explanations of many of the code constructs. This should enable you to read the example without referring to the Help. There you will find more detailed explanations of these language constructs, plus explanations of the constructs not explained here. The explanation of each piece of code follows that code.

```
const ACCEPT_TIMEOUT = 15
```

The first line of the testcase file defines the timeout value (in seconds) to be used while waiting for a window to display.

```
multitestcase MyTest (LIST OF STRING lsMachine)
```

The test case function declaration starts with the `multitestcase` keyword. It specifies a `LIST OF STRING` argument that contains the machine names for the set of client machines to be tested. You can implement and maintain this list in your test plan, by using the test plan editor. The machine names you use in this list are the names of the Agents of your target machines.

```
for each sMachine in lsMachine
  SetupMachine (sMachine, Personnel)
```

To prepare your client machines for testing, you must connect Silk Test Classic to each Agent and, by means of the Agent, bring up the application on each machine. In this example, all Agents are running the same software and so all have the same `MainWin` declaration and therefore just one test frame file. This means you can initialize all your machines the same way; for each machine being tested, you pass to `SetupMachine` the main window name you specified in your test frame file. The `SetupMachine` function issues a `Connect` call for each machine. It associates the main window name you specified (`Personnel`) with each machine so that the `DefaultBaseState` function can subsequently retrieve it.

```
SetMultiAppStates ()
```

The `SetMultiAppStates` function reads the information associated with each machine to determine whether the machine needs to be set to an application state. In this case no application state was specified (it would have been a third argument for `SetupMachine`). Therefore, `SetMultiAppStates` calls the `DefaultBaseState` function for each machine. In this example, `DefaultBaseState` drives the Agent for each machine to open the main window of the `Personnel` application. This application is then active on the client machine and `4Test` can send test case statements to the Agent to drive application operations.

```
for each sMachine in lsMachine
  spawn
    // The code to be executed in parallel by
    // all machines... (described below)
rendezvous
```

Because this is a concurrency test, you want all client applications to execute the test at exactly the same time. The `spawn` statement starts an execution thread in which each statement in the indented code block runs in parallel with all currently running threads. In this example, a thread is started for each machine in the list of machines being tested. `4Test` sends the statements in the indented code block to the Agents on each machine and then waits at the `rendezvous` statement until all Agents report that all the code statements have been executed.

The following is the code defined for the spawn statement:

```
// The code to be executed in parallel by
// all machines:
SetMachine (sMachine)
Personnel.EmployeeList.Select ("John Doe")
Personnel.Employee.Edit.Pick ()
Employee.Salary.SetText
[STRING] RandInt (50000, 70000))
```

Each thread executes operations that prepare for an attempt to perform concurrent writes to the same database record. The `SetMachine` function establishes the Agent that is to execute the code in this thread. The next two statements drive the application's user interface to select John Doe's record from the employee list box and then to pick the **Edit** option from the **Employee** menu. This opens the **Employee** dialog box and displays John Doe's employee record. The last thread operation sets the salary field in this dialog box to a random number. At this point the client is prepared to attempt a write to John Doe's employee record. When this point has been reached by all clients, the rendezvous statement is satisfied, and `4Test` can continue with the next script statement.

```
for each sMachine in lsMachine
  spawn
    SetMachine (sMachine)
    Employee.OK.Click ()
    if (MessageBox.Exists (ACCEPT_TIMEOUT))
      SetMachineData (NULL, "sMessage",
        MessageBox.Message.GetText ())
      MessageBox.OK.Click ()
      Employee.Cancel.Click ()
    else if (Employee.Exists ())
      AppError ("Employee dialog not dismissed
        after {ACCEPT_TIMEOUT} seconds")
rendezvous
```

Now that all the clients are ready to write to the database, the script creates a thread for each client, in which each attempts to save the same employee record at the same time. There is only one operation for each Agent to execute: `Employee.OK.Click`, which clicks the **OK** button to commit the edit performed in the previous thread.

The test expects the application to report the concurrency conflict with message boxes for all but one client and for that client to close its dialog box within 15 seconds. The if-else construct saves the text of the message in the error message box by means of the `SetMachineData` function. It then closes the message box and the **Employee** window so that the recovery system will not report that it had to close windows. This is good practice because it means fewer messages to interpret in the results file.

The "else if" section of the if-else checks to see whether the **Employee** window remains open, presumably because it is held by a deadlock condition; this is a test case failure. In this case, the `AppError` function places the string ******ERROR:** in front of the descriptive error message and raises an exception; all Agents terminate their threads and the test case exits.

```
iSucceed = 0
for each sMachine in lsMachine
  sMessage = GetMachineData (sMachine, "sMessage")
  if (sMessage == NULL)
    iSucceed += 1
  else
    Print ("Machine {sMachine} got message '{sMessage}'")
Verify (iSucceed, 1, "number of machines that succeeded")
```

The last section of code evaluates the results of the concurrency test in the event that all threads completed. If more than one client successfully wrote to the database, the test actually failed.

`GetMachineData` retrieves the message box message (if any) associated with each machine. If there was no message, `iSucceed` is incremented; it holds the count of "successes." The `Print` function writes the text of the message box to the results file for each machine that had a message box. You can read the results

file to verify that the correct message was reported. Alternatively, you could modify the test to automatically verify the message text.

The `Verify` function verifies that one and only one machine succeeded. If the comparison in the `Verify` function fails, `Verify` raises an exception. All exceptions are listed in the results file.

Code for `template.t`

This fragment of an example test case shows the required code with which you start a multi-application test case. It connects Silk Test Classic to all the machines being tested and brings each to its first screen. This is just a template; you must tailor your code to fit your actual needs. For information on the significance of each line of code, see *Template.t Explained*.

```
multitestcase MyTest (STRING sMach1, STRING sMach2)
  SetupMachine (sMach1, MyFirstApp, "MyFirstAppState")
  SetupMachine (sMach2, MySecondApp, "MySecondAppState")
  SetMultiAppStates ()
  spawn
    SetMachine (sMach1)
    // Here is placed code that drives test operations

  spawn
    SetMachine (sMach2)
    // Here is placed code that drives test operations

  rendezvous
  // "..."
```

template.t Explained

The following line of code in *Code for template.t* is the first required line in a multi-application test case file. It is the test case declaration.



Note: The code does not pass an application state as in the stand-alone environment.

```
multitestcase MyTest (STRING sMach1, STRING sMach2)
```

In the multi-application environment the arguments to your test case are names of the machines to be tested; you specify application states inside the test case. You can code the machine names arguments as you like. For example, you can pass a file name as the only argument, and then, in the test case, read the names of the machines from that file. Or you can define a LIST OF HMACHINE data structure in your test plan, if you are using the test plan editor, to specify the required machines and pass the name of the list, when you invoke the test case from the test plan. This template assumes that you are using a test plan and that it passes the Agent names when it invokes the test case. For this example, the test plan might specify the following:

```
Mytest ("Client1", "Client2")
```

The next two code lines are the first required lines in the test case:

```
SetupMachine (sMach1, MyFirstApp, "MyFirstAppState")
SetupMachine (sMach2, My2ndApp, "My2ndAppState")
```

You must execute the `SetupMachine` function for every client machine that will be tested. For each `SetupMachine` call, you specify the application to be tested, by passing the name of the main window, and the state to which you want the application to be set, by passing the name of the application state if you have defined one.

The `SetupMachine` function issues a `Connect` call for a machine you want to test and then configures either the base state or a specified application state.

It does this as follows:

- It associates the client application's main window name with the specified machine so that the `DefaultBaseState` function can subsequently retrieve it to set the base state.
- It associates the name of the application's base state, if one is specified, with the specified machine so that the `SetMultiAppStates` function can subsequently retrieve it and set the application to that state at the start of the test case.

The first argument for `SetUpMachine` is the machine name of one of your client machines. The second argument is the name you supply in your main window declaration in your test frame file, `frame.inc`. For this example, the `frame.inc` file specifies the following:

```
window MainWin MyFirstApp
```

Because this template specifies two different applications, it requires two different test frame files.

The third argument is the name you provide for your application state function in your appstate declaration for this test. For this example, the appstate declaration is the following:

```
appstate MyFirstAppState () based on MyFirstBaseState
```

The appstate declaration could also be of the form:

```
appstate MyFirstBaseState ()
```

Although the `DefaultBaseState` function is designed to handle most types of GUI-based applications, you may find that you need to define your own base state. It would be the application state that all your tests for this application use. In this case, you would still pass this application state to `SetUpMachine` so that your application would always be brought to this state at the start of each test case.

This template specifies two application states for generality. You would not use an application state if you wanted to start from the main window each time. If you have a number of tests that require you to bring the application to the same state, it saves test-case code to record the application state once, and pass its name to `SetUpMachine`. You will probably place your application state declarations in your test frame file.

```
SetMultiAppStates ()
```

The `SetMultiAppStates` function must always be called, even if the test case specifies no application state, because `SetMultiAppStates` calls the `DefaultBaseState` function in the absence of an appstate declaration. `SetMultiAppStates` uses the information that `SetUpMachine` associated with each connected machine to set potentially different application states or base states for each machine.

```
spawn
  SetMachine (sMach1)
  // Here is placed code that drives test operations
```

The `spawn` statement starts an execution thread, in which each statement in the indented code block below it runs in parallel with all currently running threads. There is no requirement that your test case should drive all your test machines at the same time, however, this is usually the case. The `SetMachine` function directs 4Test to execute this thread's code by means of the Agent on the specified machine. This thread can then go on to drive a portion, or all, of the test operations for this machine.

```
spawn
  SetMachine (sMach2)
  // Here is placed code that drives test operations
rendezvous
// "..."
```

The second `spawn` statement starts the thread for the second machine in this template. The `rendezvous` statement blocks the execution of the calling thread until all threads spawned have completed. You can use the `rendezvous` statement to synchronize machines as necessary before continuing with the test case.

defaults.inc

The `defaults.inc` file is provided by Silk Test Classic and implements the recovery system for a single application test. That is, it contains the `DefaultBaseState` function that performs any cleanup needed after an operation under test fails and returns the application to its base state.

You can define a base state function to replace the `DefaultBaseState` function by defining an application state without using the `basedon` keyword. This creates an application state that 4Test executes instead of the `DefaultBaseState` function.

The `defaults.inc` file contains six other functions that 4Test automatically executes unless you define functions that replace them:

DefaultScriptEnter	A null function, allows you to define a <code>ScriptEnter</code> function, as discussed below.
DefaultScriptExit (BOOLEAN bException)	Logs an exception to the results file when a script exits because of an exception.
DefaultTestcaseEnter	Executes the <code>SetAppState</code> function. If you have specified an application state for this test case, the <code>SetAppState</code> function brings your test application to that state. If you have no application state defined, <code>SetAppState</code> brings the application to the base state (if necessary).
DefaultTestcaseExit (BOOLEAN bException)	Logs an exception to the results file when a test case exits because of an exception. The function then executes the <code>SetBaseState</code> function, which calls a base state function that you have defined or the <code>DefaultBaseState</code> function.
DefaultTestPlanEnter	A null function, allows you to define <code>TestPlanEnter</code> , as discussed below, to allow logging of results.
DefaultTestPlanExit (BOOLEAN bException)	A null function, allows you to define <code>TestPlanExit</code> , as discussed below, to allow logging of results.

The word "Default" in each of the above function names signifies that you can define alternative functions to replace these. If, for example, you define a function called `TestcaseEnter`, 4Test will invoke your function before executing any of the code in your test case and will not invoke `DefaultTestcaseEnter`.

`TestPlanEnter()` is not called until the first test case in the plan is run. Or the first marked test case, if you are only running marked test cases. Similarly, `TestPlanExit()` is called at the completion of the last marked test case. `TestPlanExit()` is only called if the last marked test description contains an executable test case, which means not a manual test case or a commented out test case specifier.

cs.inc

`cs.inc` is an automatically included file that contains functions used only in the multi-application environment. The following functions provide a recovery system for managing automated testing of client/server applications:

SetMultiAppStates	Sets an application state for each connected machine, if the "AppState" machine data lists one; if not, it calls the <code>DefaultBaseState</code> function, which sets the application to its main window.
SetMultiBaseStates	Sets the application to the lowest state in the application state hierarchy for each connected machine, if the "AppState" machine data lists an application state. The lowest application state is one in which the <code>appstate</code> declaration did not use the <code>basedon</code> keyword. If there is no "AppState" information associated with this machine, <code>SetMultiBaseStates</code> calls the <code>DefaultBaseState</code> function, which sets the application to its main window, invoking it beforehand if necessary.
SetUpMachine	Connects Silk Test Classic to an agent on the specified machine. It provides a way to associate a main window declaration and an application state function with a machine name. These parameters are stored as data accessible by means of the <code>GetMachineData</code> function. Both of these

names (the second and third arguments to the function) are optional; however, if you omit both arguments, you will have no recovery system.

DefaultMultiTestCaseEnter Executes at the beginning of a multi-test case. It invokes a `DisconnectAll` function. The invocation of the `SetAppState` function is performed by the `SetMultiAppStates` function because the `DefaultTestCaseEnter` function is not executed for a multi-test case.

DefaultMultiTestCaseExit Executes just before a multi-test case terminates. It logs any pending exception, then invokes `SetMultiBaseStates` and `DisconnectAll`.

Include File Size

The maximum size of an include file is approximately 65536 lines. If your include file is very large, split it into two files and continue with your testing.

Troubleshooting Distributed Testing

This section provides troubleshooting information for testing on multiple machines.

Handling Limited Licenses

By default, Silk Test Classic starts up an unplanned Agent on the local workstation. If you do not want to use the local workstation as a test machine, set the **Agent Name** field in the **Runtime Options** dialog box to `(none)` instead of `(local)`. This will free up one license for a remote Agent.

Testing Apache Flex Applications

Silk Test provides built-in support for testing Apache Flex applications. Silk Test also provides several sample Apache Flex applications. You can access the sample applications at <http://demo.borland.com/flex/SilkTest17.5/index.html>.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Before you can test your own Apache Flex application, your Apache Flex developers must perform the following steps:

- Enabling your Apache Flex application for testing
- Creating testable Apache Flex applications
- Coding Apache Flex containers
- Implementing automation support for custom controls

To test your own Apache Flex application, follow these steps:

- Configuring security settings for your local Flash Player
- Recording a test
- Playing back a test
- Customizing Apache Flex scripts
- Testing a custom Apache Flex control



Note: Loading an Apache Flex application and initializing the Flex automation framework may take some time depending on the machine on which you are testing and the complexity of your Apache Flex application. Set the Window timeout value to a higher value to enable your application to fully load.

Overview of Apache Flex Support

Silk Test Classic provides built-in support for testing Apache Flex (Flex) applications using Internet Explorer or the Standalone Flash Player, and Adobe AIR applications built with Flex 4 or later.

Silk Test Classic also supports multiple application domains in Flex 3.x and 4.x applications, which enables you to test sub-applications. Silk Test Classic recognizes each sub-application in the locator hierarchy tree as an application tree with the relevant application domain context. At the root level in the locator attribute table, Flex 4.x sub-applications use the `SparkApplication` class. Flex 3.x sub-applications use the `FlexApplication` class.

For information on the supported versions and potential known issues, refer to the *Release Notes*.

Sample Applications

To access the Silk Test Classic sample Flex applications, go to <http://demo.borland.com/flex/SilkTest17.5/index.html>.

Object Recognition

Flex applications support hierarchical object recognition and dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. You can use both recognition methods within a single test case if necessary. Use the method best suited to meet your test requirements.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Existing Flex test cases that use hierarchical object recognition or dynamic object recognition without locator keywords in an INC file are supported. You can replay these tests, but you cannot record new tests with hierarchical object recognition or dynamic object recognition without locator keywords in an INC file. However, you can manually create tests as needed. Then, replay the tests at your convenience. For instance, any test cases that you recorded with Silk Test 2008 use hierarchical object recognition. You can replay these tests in Silk Test Classic.

Supported Controls

For a complete list of the record and replay controls available for Flex testing, refer to the *Flex Class Reference* in the *4Test Language* section of the Help.

The Silk Test Classic Flex Automation SDK is based on the Automation API for Flex. The Silk Test Classic Automation SDK supports the same components in the same manner that the Automation API for Flex supports them. For instance, the `typekey` statement in the Flex Automation API does not support all keys. You can use the `input text` statement to resolve this issue. For more information about using the Flex Automation API, refer to the *Apache Flex Release Notes*.

Agent Support

When you create a Silk Test Classic Flex project, the Open Agent is assigned as the default Agent.

Configuring Security Settings for Your Local Flash Player

Before you launch an Apache Flex application, that runs as a local application, for the first time, you must configure security settings for your local Flash Player. You must modify the Adobe specific security settings to enable the local application access to the file system.

To configure the security settings for your local Flash player:

1. Open the **Flex Security Settings Page** by clicking **Flash Player Security Manager** on <http://demo.borland.com/flex/SilkTest17.5/index.html>.
2. Click **Always allow**.
3. In the **Edit Locations** menu, click **Add Location**.
4. Click **Browse for folder** and navigate to the folder where your local application is installed.
5. Click **Confirm** and then close the browser.

Configuring Flex Applications to Run in Adobe Flash Player

To run an Apache Flex application in Flash Player, one or both of the following must be true:

- The developer who creates the Flex application must compile the application as an EXE file. When a user launches the application, it will open in Flash Player. Install Windows Flash Player from <http://www.adobe.com/support/flashplayer/downloads.html>.
- The user must have Windows Flash Player Projector installed. When a user opens a Flex .SWF file, he can configure it to open in Flash Player. Windows Flash Projector is not installed when Flash Player is installed unless you install the Apache Flex developer suite. Install Windows Flash Projector from <http://www.adobe.com/support/flashplayer/downloads.html>.

1. For Microsoft Windows 7 and Microsoft Windows Server 2008 R2, configure Flash Player to run as administrator. Perform the following steps:
 - a) Right-click the Adobe Flash Player program shortcut or the `FlashPlayer.exe` file, then click **Properties**.
 - b) In the **Properties** dialog box, click the **Compatibility** tab.
 - c) Check the **Run this program as an administrator** check box and then click **OK**.
2. Start the .SWF file in Flash Player from the command prompt (cmd.exe) by typing:


```
"<Application_Install_Directory>\ApplicationName.swf"
```

By default, the `<SilkTest_Install_Directory>` is located at `Program Files\Silk\Silk Test`.

Configuring Flex Applications for Adobe Flash Player Security Restrictions

The security model in Adobe Flash Player 10 has changed from earlier versions. When you record tests that use Flash Player, recording works as expected. However, when you play back tests, unexpected results occur when high-level clicks are used in certain situations. For instance, a **File Reference** dialog box cannot be opened programmatically and when you attempt to play back this scenario, the test fails because of security restrictions.

To work around the security restrictions, you can perform a low-level click on the button that opens the dialog box. To create a low-level click, add a parameter to the `Click` method.

For example, instead of using `SparkButton::Click()`, use `SparkButton::Click(MouseButton.Left)`. A `Click()` without parameters is a high-level click and a click with parameters (such as the button) is replayed as a low-level click.

1. Record the steps that use Flash Player.
2. Navigate to the `Click` method and add a parameter.
For example, to open the **Open File** dialog box, specify:

```
SparkButton("@caption='Open File Dialog...']").Click(MouseButton.Left)
```

When you play back the test, it works as expected.

Customizing Apache Flex Scripts

You can manually customize your Flex scripts. You can insert verifications using the **Verification** wizard. Or, you can insert verifications manually using the `Verify` function on Flex object properties.

To customize Adobe Flex scripts:

1. Record a testcase for your Flex application.
2. Open the script file that you want to customize.
3. Manually type the code that you want to add.

For example, the following code adds a verification call to your script:

```
Desktop.Find("//BrowserApplication").Find("//BrowserWindow")
.Find("//FlexApplication[@caption='explorer']").Find("//
FlexButton[@caption='OK']")
.VerifyProperties({...})
```

Each Flex object has a list of properties that you can verify. For a list of the properties available for verification, review the `Flex.inc` file. To access the file, navigate to the `<SilkTest directory>\extend\Flex` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\extend\Flex\Flex.inc`.

Styles in Apache Flex Applications

For applications developed in Apache Flex 3.x, Silk Test Classic does not distinguish between styles and properties. As a result, styles are exposed as properties. However, with Apache Flex 4.x, all new Flex controls, which are prefixed with Spark, such as `SparkButton`, do not expose styles as properties. As a result, the `GetProperty()` and `GetPropertyList()` methods for Flex 4.x controls do not return styles, such as color or font size, but only properties, such as text and name.

The `GetStyle(string styleName)` method returns values of styles as a string. To find out which styles exist, refer to the Adobe Help located at http://help.adobe.com/en_US/FlashPlatform/reference/actionsript/3/package-detail.html.

If the style is not set, a `StyleNotSetException` occurs during playback.

For the Flex 3.x controls, such as `FlexTree`, you can use `GetProperty()` to retrieve styles. Or, you can use `GetStyle()`. Both the `GetProperty()` and `GetStyle()` methods work with Flex 3.x controls.

Calculating the Color Style

In Flex, the color is represented as a number. It can be calculated using the following formula:

$red * 65536 + green * 256 + blue$

Example

In this example, the `GetProperty()` and `GetStyle()` methods are used to retrieve styles:

```
Window myTree = Application.Find("//
FlexTree[@caption='myTree']")
COLOR c = {170, 179, 179}
Verify(myTree.DisabledColor, c)
Verify(myTree.GetProperty("disabledColor"), {170, 179, 179})
Verify(myTree.GetStyle("disabledColor"), "11187123")
```

The number 11187123 for the color calculates as $170 * 65536 + 179 * 256 + 179$.

Locator Attributes for Apache Flex Controls

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Silk Test Classic supports the following locator attributes for Apache Flex (Flex) controls:

automationName	The name of the application.
caption	Similar to <code>automationName</code> .
automationClassName	For example <code>FlexButton</code> .
className	The fully qualified name of the implementation class, for example <code>mx.controls.Button</code> .
automationIndex	The index of the control in the view of the <code>FlexAutomation</code> , for example <code>index:1</code> .
index	Similar to <code>automationIndex</code> but without the prefix, for example <code>1</code> .
id	The identifier of the control.
windowId	Similar to <code>id</code> .
label	The label of the control.



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards `?` and `*`.

Dynamically Invoking Apache Flex Methods

You can call methods, retrieve properties, and set properties on controls that Silk Test Classic does not expose by using the dynamic invoke feature. This feature is useful for working with custom controls and for working with controls that Silk Test Classic supports without customization.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList()` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty()` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList()` method.



Note: Most properties are read-only and cannot be set.

Supported Methods and Properties

The following methods and properties can be called:

- Methods and properties that Silk Test Classic supports for the control.
- All public methods that the Flex API defines.
- If the control is a custom control that is derived from a standard control, all methods and properties from the standard control can be called.

Supported Parameter Types

The following parameter types are supported:

All built-in Silk Test Classic types

Silk Test Classic types includes primitive types, such as boolean, int, and string, lists, and other types, such as Point.

Returned Values

The following values are returned for properties and methods that have a return value:

- The correct value for all built-in Silk Test Classic types. These types are listed in the *Supported Parameter Types* section.
- All methods that have no return value return NULL.

Example

A custom calculator control has a `Reset` method and an `Add` method, which performs an addition of two numbers. You can use the following code to call the methods directly from your tests:

```
customControl.Invoke("Reset")  
REAL sum = customControl.DynamicInvoke("Add", {1,2})
```

Testing Multiple Flex Applications on the Same Web Page

When multiple Flex applications exist on the same web page, Silk Test Classic uses the Flex application ID or the application size property to determine which application to test. If multiple applications exist on the same page, but they are different sizes, Silk Test Classic uses the size property to determine on which application to perform any actions. Silk Test Classic uses JavaScript to find the Flex application ID to determine on which application to perform any actions if:

- multiple Flex applications exist on a single web page.
- those applications are the same size.

In this situation, if JavaScript is not enabled on the browser machine, an error occurs when a script runs.

To test multiple Flex applications that are different sizes on a single web page, follow the steps in *Testing Apache Flex Applications*.

To test multiple Flex applications that are the same size on a single web page, perform the following steps:

1. Enable JavaScript.

- In Internet Explorer:
 1. Click **Tools > Internet Options**.
 2. Click the **Security** tab.
 3. Click **Custom level**.
 4. In the **Scripting** section, under **Active Scripting**, click **Enable** and click **OK**.
- In Mozilla Firefox:
 1. Choose **Tools > Options**.
 2. Click **Content** and then check the **Enable JavaScript** check box.
 3. Click **OK**.

2. Follow the steps in *Testing Apache Flex Applications*.



Note: If a frame exists on the web page and the applications are the same size, this method will not work.

Silk Test Classic provides sample applications that demonstrate multiple applications on a single web page. You can access the sample applications at <http://demo.borland.com/flex/SilkTest17.5/index.html>.

Adobe AIR Support

Silk Test Classic supports testing with Adobe AIR for applications that are compiled with the Flex 4 compiler. For details about supported versions, check the *Release Notes* for the latest information.

Silk Test provides a sample Adobe AIR application. You can access the sample application at <http://demo.borland.com/flex/SilkTest17.5/index.html> and then click the Adobe AIR application that you want to use. You can select the application with or without automation. In order to execute the AIR application, you must install the Adobe AIR Runtime.

Apache Flex Exception Values

Exception values are generated under given error conditions. Flex support defines the following set of exception values:

<code>E_FLEX_REPLAY</code>	A generic exception, which is thrown when no other known exception occurs in Flex.
<code>E_FLEX_REPLAY_EVENT</code>	An error occurred when replaying the Flex event.
<code>E_FLEX_REPLAY_METHOD</code>	An error occurred when replaying the Flex method.
<code>E_FLEX_REPLAY_READ_PROPERTY</code>	An error occurred when reading a property.
<code>E_FLEX_REPLAY_WRITE_PROPERTY</code>	An error occurred when writing a property.
<code>E_FLEX_REPLAY_STYLE_NOT_SET</code>	The style is not set to a Flex object.
<code>E_FLEX_REPLAY_SUPPORTS_TABLUAR</code>	The property used is meant for use with tabular data. However, the specified class does not support tabular data.
<code>E_FLEX_REPLAY_INVALID_FLEX_SDK_VERSION</code>	If you replay a Flex 3.x event, method, or property in a Flex 2.0 environment, this error occurs.
<code>E_VO_PROPERTY_NOT_FOUND</code>	When reading or writing a property, if the property is not defined for the object, this exception occurs.

The `E_VO_PROPERTY_NOT_FOUND` exception can also be thrown when you test Flex, but it is not limited to the Flex environment.

Overview of the Flex Select Method Using Name or Index

You can record Flex `Select` methods using the `Name` or `Index` of the control that you select. By default, Silk Test Classic records `Select` methods using the name of the control. However, you can change your environment to record `Select` events using the index for the control, or you can switch between the name and index for recording.

You can record `Select` events using the index for the following controls:

- `FlexList`
- `FlexTree`
- `FlexDataGrid`
- `FlexAdvancedDataGrid`
- `FlexOLAPDataGrid`
- `FlexComboBox`

The default setting is `ItemBasedSelection` (`Select` event), which uses the name control. To use the index, you must adapt the `AutomationEnvironment` to use the `IndexBasedSelection` (`SelectIndex` event). To

change the behavior for one of these classes, you must modify the FlexCommonControls.xml, AdvancedDataGrid.xml, or OLAPDataGrid.xml file using the following code. Those XML files are located in the <Silk Test_install_directory>\ng\agent\plugins \com.borland.fastxd.techdomain.flex.agent_< version>\config \automationEnvironment folder. Make the following adaptations in the corresponding xml file.

```
<ClassInfo Extends="FlexList" Name="FlexControlName"
EnableIndexBasedSelection="true" >
...
</ClassInfo>
```

With this adaption the IndexBasedSelection is used for recording FlexList::SelectIndex events. Setting the EnableIndexBasedSelection= to false in the code or removing the Boolean returns recording to using the name (FlexList::Select events).



Note: You must re-start your application, which automatically re-starts the Silk Test Agent, in order for these changes to become active.

Selecting an Item in the FlexDataGrid Control

You can select an item in the FlexDataGrid control using the following procedures.

If you know the index value of the FlexDataGrid item, use the SelectIndex method.

For example, type FlexDataGrid.SelectIndex(1)

1. If you know the content value of the FlexDataGrid item, use the Select method.
2. Identify the row that you want to select with the required formatted string. Items must be separated by a pipe ("|"). At least one Item must be enclosed by two stars ("**"). This identifies the item where the click will be performed.

The syntax is: FlexDataGrid.Select("**Item1* | Item2 | Item3")

The following example selects an item using the Select method (randomly).

```
[ ] LIST OF LIST OF STRING allVisibleItems
[ ] window dataGrid =
AdobeFlashPlayer9.FlexApplication0.Index0.Index1.SwfLoader.ControlsSimpleDataG
ridSwf.DataGridControlExample.Dg
[ ]
[ ] // lets get all currently visible items
[ ] allVisibleItems = dataGrid.GetValues(dataGrid.firstVisibleRow,
dataGrid.lastVisibleRow)
[ ]
[ ] // pick a random element that we want to select
[ ] integer randomRow = RandInt(dataGrid.firstVisibleRow,
dataGrid.lastVisibleRow)
[ ] LIST OF STRING randomRowItems = allVisibleItems[randomRow]
[ ] print("This is the row we want to select: {randomRow}")
[ ]
[ ] // now lets construct the string we need for the select method
[ ] STRING selectString
[ ] STRING itemText
[ ] INTEGER col = 0
[-] for each itemText in randomRowItems
[-] if col == 0
[ ] selectString = "{itemText}*"
[-] else
[ ] selectString = selectString + " | {itemText}"
[ ] col++
[ ]
```



```
[ ] // now lets select the item
[ ] print("We will select {selectString}")
[ ] dataGrid.Select(selectString)
```

Enabling Your Flex Application for Testing

To enable your Flex application for testing, your Apache Flex developers must include the following components in the Flex application:

- Apache Flex Automation Package
- Silk Test Automation Package

Apache Flex Automation Package

The Flex automation package provides developers with the ability to create Flex applications that use the Automation API. You can download the Flex automation package from Adobe's website, <http://www.adobe.com>. The package includes:

- Automation libraries – the automation.swc and automation_agent.swc libraries are the implementations of the delegates for the Flex framework components. The automation_agent.swc file and its associated resource bundle are the generic agent mechanism. An agent, such as the Silk Test Agent, builds on top of these libraries.
- Samples



Note: The Silk Test Flex Automation SDK is based on the Automation API for Flex. The Silk Test Automation SDK supports the same components in the same manner that the Automation API for Flex supports them. For instance, the `typekey` statement in the Flex Automation API does not support all keys. You can use the input text statement to resolve this issue. For more information about using the Flex Automation API, see the *Apache Flex Release Notes*.

Silk Test Automation Package

Silk Test's Open Agent uses the Apache Flex automation agent libraries. The FlexTechDomain.swc file contains the Silk Test specific implementation.

You can enable your application for testing using either of the following methods:

- Linking automation packages to your Flex application
- Run-time loading

Linking Automation Packages to Your Flex Application

You must precompile Flex applications that you plan to test. The functional testing classes are embedded in the application at compile time, and the application has no external dependencies for automated testing at run time.

When you embed functional testing classes in your application SWF file at compile time, the size of the SWF file increases. If the size of the SWF file is not important, use the same SWF file for functional testing and deployment. If the size of the SWF file is important, generate two SWF files, one with functional testing classes embedded and one without. Use the SWF file that does not include the embedded testing classes for deployment.


When you precompile the Flex application for testing, in the include-libraries compiler option, reference the following files:

- automation.swc
- automation_agent.swc
- FlexTechDomain.swc
- automation_charts.swc (include only if your application uses charts and Flex 2.0)
- automation_dmv.swc (include if your application uses charts and Flex > 3.x)

- automation_flasflexkit.swc (include if your application uses embedded flash content)
- automation_spark.swc (include if your application uses the new Flex 4.x controls)
- automation_air.swc (include if your application is an AIR application)
- automation_airsdk.swc (include if your application is an AIR application and uses new Flex 4.x controls)

When you create the final release version of your Flex application, you recompile the application without the references to these SWC files. For more information about using the automation SWC files, see the *Apache Flex Release Notes*.

If you do not deploy your application to a server, but instead request it by using the file protocol or run it from within Apache Flex Builder, you must include each SWF file in the local-trusted sandbox. This requires additional configuration information. Add the additional configuration information by modifying the compiler's configuration file or using a command-line option.


 **Note:** The Silk Test Flex Automation SDK is based on the Automation API for Flex. The Silk Test Automation SDK supports the same components in the same manner that the Automation API for Flex supports them. For instance, when an application is compiled with automation code and successive SWF files are loaded, a memory leak occurs and the application runs out of memory eventually. The Flex Control Explorer sample application is affected by this issue. The workaround is to not compile the application SWF files that Explorer loads with automation libraries. For example, compile only the Explorer main application with automation libraries. Another alternative is to use the module loader instead of swfloader. For more information about using the Flex Automation API, see the *Apache Flex Release Notes*.

Precompiling the Flex Application for Testing

You can enable your application for testing by precompiling your application for testing or by using run-time loading.

1. Include the automation.swc, automation_agent.swc, and FlexTechDomain.swc libraries in the compiler's configuration file by adding the following code to the configuration file:

```
<include-libraries>
...
<library>/libs/automation.swc</library>
<library>/libs/automation_agent.swc</library>
<library>pathinfo/FlexTechDomain.swc</library>
</include-libraries>
```

 **Note:** If your application uses charts, you must also add the automation_charts.swc file.

2. Specify the location of the automation.swc, automation_agent.swc, and FlexTechDomain.swc libraries using the include-libraries compiler option with the command-line compiler.


The configuration files are located at:


Apache Flex 2 SDK – <flex_installation_directory>/frameworks/flex-config.xml

Apache Flex Data Services – <flex_installation_directory>/flex/WEB-INF/flex/flex-config.xml

The following example adds the automation.swc and automation_agent.swc files to the application:

```
mxmlc -include-libraries+=../frameworks/libs/automation.swc;../frameworks/
libs/
automation_agent.swc;pathinfo/FlexTechDomain.swc MyApp.mxml
```

 **Note:** Explicitly setting the include-libraries option on the command line overwrites, rather than appends, the existing libraries. If you add the automation.swc and automation_agent.swc files using the include-libraries option on the command line, ensure that you use the += operator. This appends rather than overwrites the existing libraries that are included.

 **Note:** The Silk Test Flex Automation SDK is based on the Automation API for Flex. The Silk Test Automation SDK supports the same components in the same manner that the Automation API for Flex supports them. For instance, when an application is compiled with automation code and successive SWF files are loaded, a memory leak occurs and the application runs out of memory eventually. The Flex Control Explorer sample application is affected by this issue. The workaround is to not compile the application SWF files that Explorer loads with automation libraries. For example, compile only the Explorer main application with automation libraries. Another alternative is to use the module loader instead of swfloader. For more information about using the Flex Automation API, see the *Apache FlexRelease Notes*.

Loading at Run-Time

1. Copy the content of the `Silk\Silk Test\ng\AutomationSDK\Flex\<version>\FlexAutomationLauncher` directory into the directory of the Flex application that you are testing.
2. Open `FlexAutomationLauncher.html` in Windows Explorer and add the following parameter as a suffix to the file path:

```
?automationurl=YourApplication.swf
```

where *YourApplication.swf* is the name of the SWF file for your Flex application.

3. Add `file:///` as a prefix to the file path.

For example, if your file URL includes a parameter, such as: `?automationurl=explorer.swf`, type: .

```
file:///C:/Program%20Files/Silk/Silk Test/ng/sampleapplications/Flex/3.2/  
FlexControlExplorer32/FlexAutomationLauncher.html?automationurl=explorer.swf
```

Run-Time Loading

You can load Flex automation support at run time using the Silk Test Flex Automation Launcher. This application is compiled with the automation libraries and loads your application with the SWFLoader class. This automatically enables your application for testing without compiling automation libraries into your SWF file. The Silk Test Flex Automation Launcher is available in HTML and SWF file formats.

Limitations

- The Flex Automation Launcher Application automatically becomes the root application. If your application must be the root application, you cannot load automation support with the Silk Test Flex Automation Launcher.
- Testing applications that load external libraries – Applications that load other SWF file libraries require a special setting for automated testing. A library that is loaded at run time (including run-time shared libraries (RSLs) must be loaded into the ApplicationDomain of the loading application. If the SWF file used in the application is loaded in a different application domain, automated testing record and playback will not function properly. The following example shows a library that is loaded into the same ApplicationDomain:

```
import flash.display.*;  
  
import flash.net.URLRequest;  
  
import flash.system.ApplicationDomain;  
  
import flash.system.LoaderContext;
```

```

var ldr:Loader = new Loader();

var urlReq:URLRequest = new URLRequest("RuntimeClasses.swf");

var context:LoaderContext = new LoaderContext();

context.applicationDomain = ApplicationDomain.currentDomain;

loader.load(request, context);

```

Using the Command Line to Add Configuration Information

To specify the location of the `automation.swc`, `automation_agent.swc`, and `FlexTechDomain.swc` libraries using the command-line compiler, use the `include-libraries` compiler option.

The following example adds the `automation.swc` and `automation_agent.swc` files to the application:

```

mxmmlc -include-libraries+=../frameworks/libs/automation.swc;../frameworks/
libs/
automation_agent.swc;pathinfo/FlexTechDomain.swc MyApp.mxml

```



Note: If your application uses charts, you must also add the `automation_charts.swc` file to the `include-libraries` compiler option.

Explicitly setting the `include-libraries` option on the command line overwrites, rather than appends, the existing libraries. If you add the `automation.swc` and `automation_agent.swc` files using the `include-libraries` option on the command line, ensure that you use the `+=` operator. This appends rather than overwrites the existing libraries that are included.

To add automated testing support to a Flex Builder project, you must also add the `automation.swc` and `automation_agent.swc` files to the `include-libraries` compiler option.

Passing Parameters into a Flex Application

You can pass parameters into a Flex application using the following procedures.

Passing Parameters into a Flex Application Before Runtime

You can pass parameters into a Flex application before runtime using automation libraries.

1. Compile your application with the appropriate automation libraries.
2. Use the standard Flex mechanism for the parameter as you typically would.

Passing Parameters into a Flex Application at Runtime Using the Flex Automation Launcher

Before you begin this task, prepare your application for run-time loading.

1. Open the `FlexAutomationLauncher.html` file or create a file using `FlexAutomationLauncher.html` as an example.
2. Navigate to the following section:

```

<script language="JavaScript" type="text/javascript">

    AC_FL_RunContent(eef

        "src", "FlexAutomationLauncher",

        "width", "100%",

        "height", "100%",


```

```

        "align", "middle",
        "id", "FlexAutomationLauncher",
        "quality", "high",
        "bgcolor", "white",
        "name", "FlexAutomationLauncher",
        "allowScriptAccess", "sameDomain",
        "type", "application/x-shockwave-flash",
        "pluginspage", "http://www.adobe.com/go/getflashplayer",
        "flashvars", "yourParameter=yourParameterValue"+
"&automationurl=YourApplication.swf"

    );
</script>

```

 **Note:** Do not change the "FlexAutomationLauncher" value for "src", "id", or "name."

3. Add your own parameter to "*yourParameter=yourParameterValue*".
4. Pass the name of the Flex application that you want to test as value for the "&*automationurl=YourApplication.swf*" value.
5. Save the file.

Creating Testable Flex Applications

As a Flex developer, you can employ techniques to make Flex applications as "test friendly" as possible. These include:

- Providing Meaningful Identification of Objects
- Avoiding Duplication of Objects

Providing Meaningful Identification of Objects

To create "test friendly" applications, ensure that objects are identifiable in scripts. You can set the value of the ID property for all controls that are tested, and ensure that you use a meaningful string for that ID property.

To provide meaningful identification of objects:

- Give all testable MXML components an ID to ensure that the test script has a unique identifier to use when referring to that Flex control.
- Make these identifiers as human-readable as possible to make it easier for the user to identify that object in the testing script. For example, set the id property of a Panel container inside a TabNavigator to `submit_panel` rather than `panel1` or `p1`.

When working with Silk Test Classic, an object is automatically given a name depending on certain tags, for instance, `id`, `childIndex`. If there is no value for the `id` property, Silk Test Classic uses other properties, such as the `childIndex` property. Assigning a value to the `id` property makes the testing scripts easier to read.

Avoiding Duplication of Objects

Automation agents rely on the fact that some properties of object instances will not be changed during run time. If you change the Flex component property that is used by Silk Test Classic as the object name at run time, unexpected results can occur. For example, if you create a Button control without an `automationName` property, and you do not initially set the value of its label property, and then later set the value of the `label` property, problems might occur. In this case, Silk Test Classic uses the value of the label property of Button controls to identify an object if the `automationName` property is not set. If you later set the value of the `label` property, or change the value of an existing label, Silk Test Classic identifies the object as a new object and does not reference the existing object.

To avoid duplicating objects:

- Understand what properties are used to identify objects in the agent and avoid changing those properties at run time.
- Set unique, human-readable `id` or `automationName` properties for all objects that are included in the recorded script.

Flex AutomationName and AutomationIndex Properties

The Flex Automation API introduces the `automationName` and `automationIndex` properties. If you provide the `automationName`, Silk Test Classic uses this value for the recorded window declaration's name. Providing a meaningful name makes it easier for Silk Test Classic to identify that object. As a best practice, set the value of the `automationName` property for all objects that are part of the application's test.

Use the `automationIndex` property to assign a unique index value to an object. For instance, if two objects share the same name, assign an index value to distinguish between the two objects.



Note: The Silk Test Flex Automation SDK is based on the Automation API for Flex. The Silk Test Automation SDK supports the same components in the same manner that the Automation API for Flex supports them. For instance, when an application is compiled with automation code and successive SWF files are loaded, a memory leak occurs and the application runs out of memory eventually. The Flex Control Explorer sample application is affected by this issue. The workaround is to not compile the application SWF files that Explorer loads with automation libraries. For example, compile only the Explorer main application with automation libraries. Another alternative is to use the module loader instead of `swfloader`. For more information about using the Flex Automation API, see the *Apache Flex Release Notes*.

Setting the Flex automationName Property

The `automationName` property defines the name of a component as it appears in tests. The default value of this property varies depending on the type of component. For example, the `automationName` for a Button control is the label of the Button control. Sometimes, the `automationName` is the same as the `id` property for the control, but this is not always the case.

For some components, Flex sets the value of the `automationName` property to a recognizable attribute of that component. This helps testers recognize the component in their tests. Because testers typically do not have access to the underlying source code of the application, having a control's visible property define that control can be useful. For example, a Button labeled "Process Form Now" appears in the test as `FlexButton("Process Form Now")`.

If you implement a new component, or derive from an existing component, you might want to override the default value of the `automationName` property. For example, `UIComponent` sets the value of the `automationName` to the component's `id` property by default. However, some components use their own methods for setting the value. For example, in the Flex Store sample application, containers are used to create the product thumbnails. A container's default `automationName` would not be very useful because it is the same as the container's `id` property. So, in Flex Store, the custom component that generates a product thumbnail explicitly sets the `automationName` to the product name to make testing the application easier.

Example

The following example from the `CatalogPanel.mxml` custom component sets the value of the `automationName` property to the name of the item as it appears in the catalog. This is more recognizable than the default automation name.

```
thumbs[i].automationName = catalog[i].name;
```

Example

The following example sets the `automationName` property of the `ComboBox` control to "Credit Card List"; rather than using the `id` property, the testing tool typically uses "Credit Card List" to identify the `ComboBox` in its scripts:

```
<?xml version="1.0"?>
<!-- at/SimpleComboBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      [Bindable]
      public var cards: Array = [
        {label:"Visa", data:1},
        {label:"MasterCard", data:2},
        {label:"American Express", data:3}
      ];

      [Bindable]
      public var selectedItem:Object;
    ]
  >
</mx:Script>
<mx:Panel title="ComboBox Control Example">
  <mx:ComboBox id="cb1" dataProvider="{cards}"
    width="150"
    close="selectedItem=ComboBox(event.target).selectedItem"
    automationName="Credit Card List"
  />
  <mx:VBox width="250">
    <mx:Text width="200" color="blue" text="Select a type of
credit card." />
    <mx:Label text="You selected: {selectedItem.label}"/>
    <mx:Label text="Data: {selectedItem.data}"/>
  </mx:VBox>
</mx:Panel>
</mx:Application>
```

Setting the value of the `automationName` property ensures that the object name will not change at run time. This helps to eliminate unexpected results.

If you set the value of the `automationName` property, tests use that value rather than the default value. For example, by default, Silk Test Classic uses a `Button` control's `label` property as the name of the `Button` in the script. If the label changes, the script can break. You can prevent this from happening by explicitly setting the value of the `automationName` property.

Buttons that have no label, but have an icon, are recorded by their index number. In this case, ensure that you set the `automationName` property to something meaningful so that the tester can recognize the `Button` in the script. After the value of the `automationName` property is set, do not change the value during the component's life cycle. For item renderers, use the `automationValue` property rather than the `automationName` property. To use the `automationValue` property, override the

`createAutomationIDPart()` method and return a new value that you assign to the `automationName` property, as the following example shows:

```
<mx:List xmlns:mx="http://www.adobe.com/2006/mxml" >
  <mx:Script>
    <![CDATA[
      import mx.automation.IAutomationObject;
      override public function
      createAutomationIDPart(item:IAutomationObject):Object {
        var id:Object = super.createAutomationIDPart(item);
        id["automationName"] = id["automationIndex"];
        return id;
      }
    ]]>
  </mx:Script>
</mx:List>
```

Use this technique to add index values to the children of any container or list-like control. There is no method for a child to specify an index for itself.

Setting the Flex Select Method to Use Name or Index

You can record Flex Select methods using the Name or Index of the control that you select. By default, Silk Test records Select methods using the name of the control. However, you can change your environment to record Select events using the index for the control, or you can switch between the name and index for recording.

1. Determine which class you want to modify to use the Index.

You can record Select events using the index for the following controls:

- FlexList
- FlexTree
- FlexDataGrid
- FlexOLAPDataGrid
- FlexComboBox
- FlexAdvancedDataGrid

2. Determine which XML file is related to the class that you want to modify.

The XML files related to the preceding controls include: `FlexCommonControls.xml`, `AdvancedDataGrid.xml`, or `OLAPDataGrid.xml`.

3. Navigate to the XML files that are related to the class that you want to modify.

The XML files are located in the `<Silk Test_install_directory>\ng\agent\plugins\com.borland.fastxd.techdomain.flex.agent_<version>\config\automationEnvironment` folder.

4. Make the following adaptations in the corresponding XML file.

```
<ClassInfo Extends="FlexList" Name="FlexControlName"
EnableIndexBasedSelection="true" >
...
</ClassInfo>
```

For instance, you might use "FlexList" as the "FlexControlName" and modify the `FlexCommonControls.xml` file.

With this adaption the `IndexBasedSelection` is used for recording `FlexList::SelectIndex` events.



Note: Setting the `EnableIndexBasedSelection` to `false` in the code or removing the boolean returns recording to using the name (`FlexList::Select` events).

5. Re-start your Flex application and the Open Agent in order for these changes to become active.

Coding Flex Containers

Containers differ from other kinds of controls because they are used both to record user interactions (such as when a user moves to the next pane in an Accordion container) and to provide unique locations for controls in the testing scripts.

Adding and Removing Containers from the Automation Hierarchy

In general, the automated testing feature reduces the amount of detail about nested containers in its scripts. It removes containers that have no impact on the results of the test or on the identification of the controls from the script. This applies to containers that are used exclusively for layout, such as the `HBox`, `VBox`, and `Canvas` containers, except when they are being used in multiple-view navigator containers, such as the `ViewStack`, `TabNavigator`, or `Accordion` containers. In these cases, they are added to the automation hierarchy to provide navigation.

Many composite components use containers, such as `Canvas` or `VBox`, to organize their children. These containers do not have any visible impact on the application. As a result, you typically exclude these containers from testing because there is no user interaction and no visual need for their operations to be recordable. By excluding a container from testing, the related test script is less cluttered and easier to read.

To exclude a container from being recorded (but not exclude its children), set the container's `showInAutomationHierarchy` property to `false`. This property is defined by the `UIComponent` class, so all containers that are a subclass of `UIComponent` have this property. Children of containers that are not visible in the hierarchy appear as children of the next highest visible parent.

The default value of the `showInAutomationHierarchy` property depends on the type of container. For containers such as `Panel`, `Accordion`, `Application`, `DividedBox`, and `Form`, the default value is `true`; for other containers, such as `Canvas`, `HBox`, `VBox`, and `FormItem`, the default value is `false`.

The following example forces the `VBox` containers to be included in the test script's hierarchy:

```
<?xml version="1.0"?>
<!-- at/NestedButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
<mx:Panel title="ComboBox Control Example">
<mx:HBox id="hb">
<mx:VBox id="vb1" showInAutomationHierarchy="true">
<mx:Canvas id="c1">
<mx:Button id="b1" automationName="Nested Button 1" label="Click Me" />
</mx:Canvas>
</mx:VBox>
<mx:VBox id="vb2" showInAutomationHierarchy="true">
<mx:Canvas id="c2">
<mx:Button id="b2" automationName="Nested Button 2" label="Click Me 2" />
</mx:Canvas>
</mx:VBox>
</mx:HBox>
</mx:Panel>
</mx:Application>
```

Multiview Containers

Avoid using the same label on multiple tabs in multiview containers, such as the `TabNavigator` and `Accordion` containers. Although it is possible to use the same labels, this is generally not an acceptable UI design practice and can cause problems with control identification in your testing environment.

Flex Automation Testing Workflow

The Silk Test Classic workflow for testing Flex applications includes:

- Automated Testing Initialization
- Automated Testing Recording
- Automated Testing Playback

Flex Automated Testing Initialization

When the user launches the Flex application, the following initialization events occur:

1. The automation initialization code associates component delegate classes with component classes.
2. The component delegate classes implement the `IAutomationObject` interface.
3. An instance for the `AutomationManager` is created in the mixin `init()` method. (The `AutomationManager` is a mixin.)
4. The `SystemManager` initializes the application. Component instances and their corresponding delegate instances are created. Delegate instances add event listeners for events of interest.
5. The Silk Test Classic `FlexTechDomain` is a mixin. In the `FlexTechDomain init()` method, the `FlexTechDomain` registers for the `SystemManager.APPLICATION_COMPLETE` event. When the event is received, it creates a `FlexTechDomain` instance.
6. The `FlexTechDomain` instance connects via a TCP/IP socket to the Silk Test Agent on the same machine that registers for record/playback functionality.
7. The `FlexTechDomain` requests information about the automation environment. This information is stored in XML files and is forwarded from the Silk Test Agent to the `FlexTechDomain`.

Flex Automated Testing Recording

When the user records a new test in Silk Test Classic for a Flex application, the following events occur:

1. Silk Test Classic calls the Silk Test Agent to start recording. The Agent forwards this command to the `FlexTechDomain` instance.
2. `FlexTechDomain` notifies `AutomationManager` to start recording by calling `beginRecording()`. The `AutomationManager` adds a listener for the `AutomationRecordEvent.RECORD` event from the `SystemManager`.
3. The user interacts with the application. For example, suppose the user clicks a `Button` control.
4. The `ButtonDelegate.clickEventHandler()` method dispatches an `AutomationRecordEvent` event with the click event and `Button` instance as properties.
5. The `AutomationManager` record event handler determines which properties of the click event to store based on the XML environment information. It converts the values into proper type or format. It dispatches the record event.
6. The `FlexTechDomain` event handler receives the event. It calls the `AutomationManager.createID()` method to create the `AutomationID` object of the button. This object provides a structure for object identification. The `AutomationID` structure is an array of `AutomationIDParts`. An `AutomationIDPart` is created by using `IAutomationObject`. (The `UIComponent.id`, `automationName`, `automationValue`, `childIndex`, and `label` properties of the `Button` control are read and stored in the object. The `label` property is used because the XML information specifies that this property can be used for identification for the `Button`.)
7. `FlexTechDomain` uses the `AutomationManager.getParent()` method to get the logical parent of `Button`. The `AutomationIDPart` objects of parent controls are collected at each level up to the application level.
8. All the `AutomationIDParts` are included as part of the `AutomationID` object.
9. The `FlexTechDomain` sends the information in a call to Silk Test Classic.
10. When the user stops recording, the `FlexTechDomain.endRecording()` method is called.

Flex Automated Testing Playback

When the user clicks the **Playback** button in Silk Test Classic, the following events occur:

1. For each script call, Silk Test Classic contacts the Silk Test Agent and sends the information for the script call to be executed. This information includes the complete window declaration, the event name, and parameters.
2. The Silk Test Agent forwards that information to the FlexTechDomain.
3. The FlexTechDomain uses `AutomationManager.resolveIDToSingleObject` with the window declaration information. The AutomationManager returns the resolved object based on the descriptive information (automationName, automationIndex, id, and so on).
4. Once the Flex control is resolved, FlexTechDomain calls `AutomationManager.replayAutomatableEvent()` to replay the event.
5. The `AutomationManager.replayAutomatableEvent()` method invokes the `IAutomationObject.replayAutomatableEvent()` method on the delegate class. The delegate uses the `IAutomationObjectHelper.replayMouseEvent()` method (or one of the other replay methods, such as `replayKeyboardEvent()`) to play back the event.
6. If there are verifications in your script, FlexTechDomain invokes `AutomationManager.getProperties()` to access the values that must be verified.

Testing the Component Explorer

Silk Test provides a sample Apache Flex test application called the **Component Explorer**. You can access the **Component Explorer** at http://demo.borland.com/flex/SilkTest17.5/3.5/Flex3TestApp_withAutomation/Flex3TestApp.html. The topics in this section describe how you can test the **Component Explorer**.

Silk Test provides several additional Apache Flex sample applications. To access the sample applications, go to <http://demo.borland.com/flex/SilkTest17.5/index.html>.

Configuring Security Settings for Your Local Flash Player

Before you launch an Apache Flex application, that runs as a local application, for the first time, you must configure security settings for your local Flash Player. You must modify the Adobe specific security settings to enable the local application access to the file system.

To configure the security settings for your local Flash player:

1. Open the **Flex Security Settings Page** by clicking **Flash Player Security Manager** on <http://demo.borland.com/flex/SilkTest17.5/index.html>.
2. Click **Always allow**.
3. In the **Edit Locations** menu, click **Add Location**.
4. Click **Browse for folder** and navigate to the folder where your local application is installed.
5. Click **Confirm** and then close the browser.

Launching the Component Explorer

Silk Test provides a sample Apache Flex application, the Component Explorer. Compiled with the Adobe Automation SDK and the Silk Test specific automation implementation, the Component Explorer is pre-configured for testing.

Before you launch the application for the first time, you must configure security settings for your local Flash Player.

To launch the Component Explorer in Internet Explorer, open http://demo.borland.com/flex/SilkTest17.5/3.5/Flex3TestApp_withAutomation/Flex3TestApp.html.

The application launches in Internet Explorer.

Creating a New Project

You can create a new project and add the appropriate files to the project, or you can have Silk Test Classic automatically create a new project from an existing file.

Since each project is a unique testing environment, by default new projects do not contain any settings, such as extensions, class mappings, or Agent options. If you want to retain the settings from your current test set, save them as an options set by opening Silk Test Classic and clicking **Options > Save New Options Set**. You can add the options set to your project.

To create a new project:

1. In Silk Test Classic, click **File > New Project**, or click **Open Project > New Project** on the basic workflow bar. the **Create Project** dialog box appears.
2. Type a unique name for the project into the **Project Name** field.
If the project name already exists, you are prompted to specify a different project name or overwrite the existing project.
Project files with a `.vtp` (Verify Test Project) extension, `projectname.vtp`, and a `projectname.ini` are created in the directory that you specify in the **Save in** text box.
3. *Optional:* Type a description for the project into the **Description** field.
After you create the project, you can view the project description by right-clicking the `projectname` folder on the **Files** or **Global** tabs or clicking **Project > Project Description**.
4. Type the location at which you want to save the project into the **Save in** field.
You can also click **Browse** and specify the folder in which you want to save your project. The default location is `C:\Users\\Documents\Silk Test Classic Projects`.
5. Click **OK**. Silk Test Classic creates a `<Project name>` folder within the specified directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension `.ini` files, which are `appexpx.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project. Silk Test Classic then creates your project and displays nodes on the **Files** and **Global** tabs for the files and resources associated with this project.
6. Perform one of the following steps:
 - If your test uses the Open Agent, configure the application to set up the test environment.
 - If your test uses the Classic Agent, enable the appropriate extensions to test your application.

Configuring Web Applications

This functionality is supported only if you are using the Open Agent.

Configure the web application that you want to test to set up the environment that Silk Test Classic will create each time you record or replay a test case. If you are testing a web application or an application that uses a child technology domain of the xBrowser technology domain, for example an Apache Flex application, use this configuration.

1. Click **Configure Applications** on the basic workflow bar.
If you do not see **Configure Applications** on the workflow bar, ensure that the default agent is set to the Open Agent.
The **Select Application** dialog box opens.
2. Select the **Web** tab.
3. Select the browser that you want to use from the list of available browsers.
If you want to record a test against a web application, select a browser that is supported for recording. To see which browsers are supported for recording, refer to the [Release Notes](#).

4. Specify the web page to open in the **Enter URL to navigate** text box or click **Use URL from running browser**.
5. *Optional:* Check the **Create Base State** check box to create a base state for the application under test. By default, the **Create Base State** check box is checked for projects where a base state for the application under test is not defined, and unchecked for projects where a base state is defined. An application's base state is the known, stable state that you expect the application to be in before each test begins execution, and the state the application can be returned to after each test has ended execution. When you configure an application and create a base state, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box.
6. Click **OK**.
 - If you have checked the **Create Base State** check box, the **Choose name and folder of the new frame file** page opens. Silk Test Classic configures the recovery system and names the corresponding file `frame.inc` by default.
 - If you have not checked the **Create Base State** check box, the dialog box closes and you can skip the remaining steps.
7. Navigate to the location in which you want to save the frame file.
8. In the **File name** text box, type the name for the frame file that contains the default base state and recovery system. Then, click **Save**. Silk Test Classic creates a base state for the application. By default, Silk Test Classic lists the caption of the main window of the application as the locator for the base state. Then Silk Test Classic opens the Web page.
9. Record the test case whenever you are ready.

Recording a Sample Test Case for the Component Explorer

Use the following procedure to become familiar with the sample Silk Test Classic Flex application, the Component Explorer.

To record a test case for the Component Explorer:

1. Click **Record Testcase** on the Basic Workflow bar.
2. In the **Record Testcase** dialog box, type the name of your test case into the **Testcase name** field. Test case names are case sensitive; they can have any length and consist of any combination of alphabetic characters, numerals, and underscore characters.
3. From the **Application State** list box, select **DefaultBaseState** to have the built-in recovery system restore the default BaseState before the test case begins executing.
4. Click **Start Recording**. Silk Test Classic closes the **Record Testcase** dialog box and displays the Flex sample application.
5. When the **Recording** window opens, record the following scenario using the Flex sample application. It is essential that you perform these steps exactly as they are documented. Otherwise, your test case script may not match the sample provided later in this document.
6. Click the ► arrow next to the **Visual Components** tree element to expand the list.
7. Click the ► arrow next to the **General Controls** tree element to expand the list.
8. Click the **SimpleAlert** tree element.
9. In the **Alert Control Example** section, click **Click Me** near the top of the window and then click **OK** in the **Hello World** message box.
10. Click the ▼ arrow next to the **General Controls** tree element to hide the list.
11. Click the ▼ arrow next to the **Visual Components** tree element to hide the list.
12. In the **Recording** window, click **Stop**. Silk Test Classic opens the **Record Test Case** dialog box, which contains the recorded script.

13. Click **Paste to Editor**. The **Update Files** dialog box opens.

14. Choose **Paste test case and update window declaration(s)** and then click **OK**.

Your test case should include the following calls:

```
WebBrowser.BrowserWindow.Application.CompLibTree.Open("Visual Components")
WebBrowser.BrowserWindow.Application.CompLibTree.Open("Visual
Components>General Controls")
WebBrowser.BrowserWindow.Application.CompLibTree.Select("Visual
Components>General Controls>SimpleAlert")
WebBrowser.BrowserWindow.Application.Button1.Click()
WebBrowser.BrowserWindow.Application.Ok.Click()
WebBrowser.BrowserWindow.Application.CompLibTree.Close("Visual
Components>General Controls")
WebBrowser.BrowserWindow.Application.CompLibTree.Close("Visual Components")
```

The Silk Test Classic Flex Automation SDK is based on the Automation API for Flex. The Silk Test Classic Automation SDK supports the same components in the same manner that the Automation API for Flex supports them. For instance, when an application is compiled with automation code and successive `.swf` files are loaded, a memory leak occurs and the application runs out of memory eventually. The Flex Component Explorer sample application is affected by this issue. The workaround is to not compile the application `.swf` files that Explorer loads with automation libraries. For example, compile only the Explorer main application with automation libraries. Another alternative is to use the module loader instead of `swfloader`. For more information about using the Flex Automation API, refer to the *Apache Flex Release Notes*.

Running a Test Case

When you run a test case, Silk Test Classic interacts with the application by executing all the actions you specified in the test case and testing whether all the features of the application performed as expected.

Silk Test Classic always saves the suite, script, or test plan before running it if you made any changes to it since the last time you saved it. By default, Silk Test Classic also saves all other open modified files whenever you run a script, suite, or test plan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog box.

1. Make sure that the test case that you want to run is in the active window.
2. Click **Run Testcase** on the **Basic Workflow** bar.
If the workflow bar is not visible, choose **Workflows > Basic** to enable it.
Silk Test Classic displays the **Run Testcase** dialog box, which lists all the test cases contained in the current script.
3. Select a test case and specify arguments, if necessary, in the **Arguments** field.
Remember to separate multiple arguments with commas.

4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

- BaseStateExecutionFinished
- Connecting
- Verify
- Exists
- Is
- Get
- Set
- Print

- ForceActiveXEnum
 - Wait
 - Sleep
5. To view results using the TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.
 6. Click **Run**. Silk Test Classic runs the test case and generates a results file.
For the Classic Agent, multiple tags are supported. If you are running test cases using other agents, you can run scripts that use declarations with multiple tags. To do this, check the **Disable Multiple Tag Feature** check box in the **Agent Options** dialog box on the **Compatibility** tab. When you turn off multiple-tag support, 4Test discards all segments of a multiple tag except the first one.
 7. *Optional:* If necessary, you can press both **Shift** keys at the same time to stop the execution of the test.

Customizing Apache Flex Scripts

You can manually customize your Flex scripts. You can insert verifications using the **Verification** wizard. Or, you can insert verifications manually using the `Verify` function on Flex object properties.

To customize Adobe Flex scripts:

1. Record a testcase for your Flex application.
2. Open the script file that you want to customize.
3. Manually type the code that you want to add.

For example, the following code adds a verification call to your script:

```
Desktop.Find("//BrowserApplication").Find("//BrowserWindow")
.Find("//FlexApplication[@caption='explorer']").Find("//
FlexButton[@caption='OK']")
.VerifyProperties({...})
```

Each Flex object has a list of properties that you can verify. For a list of the properties available for verification, review the `Flex.inc` file. To access the file, navigate to the `<SilkTest directory>\extend\Flex` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\extend\Flex\Flex.inc`.

Testing Flex Custom Controls

Silk Test Classic supports testing Flex custom controls. By default, Silk Test Classic provides record and playback support for the individual sub-controls of the custom control.

For testing custom controls, the following options exist:

Option	Description
Basic support	With basic support, you use dynamic invoke to interact with the custom control during replay. Use this low-effort approach when you want to access properties and methods of the custom control in the test application that Silk Test Classic does not expose. The developer of the custom control can also add methods and properties to the custom control specifically for making the control easier to test. A Silk Test Classic user can then call those methods or properties using the dynamic invoke feature.

The advantages of basic support include:

- Dynamic invoke requires no code changes in the test application.
- Using dynamic invoke is sufficient for most testing needs.

The disadvantages of basic support include:

Option Description

- No specific class name is included in the locator. For example, Silk Test Classic records `//FlexBox` rather than `//FlexSpinner`.
- Only limited recording support.
- Silk Test Classic cannot replay events.

For more details about dynamic invoke, including an example, see *Dynamically Invoking Apache Flex Methods*.

Advanced support

With advanced support, you create specific automation support for the custom control. This additional automation support provides recording support and more powerful play-back support. The advantages of advanced support include:

- High-level recording and playback support, including the recording and replaying of events.
- Silk Test Classic treats the custom control exactly the same as any other built-in Flex control.
- Seamless integration into Silk Test Classic API.
- Silk Test Classic uses the specific class name in the locator. For example, Silk Test Classic records `//FlexSpinner`.

The disadvantages of advanced support include:

- Implementation effort is required. The test application must be modified and the Open Agent must be extended.

Defining a Custom Control in the Test Application

Typically, the test application already contains custom controls, which were added during development of the application. If your test application already includes custom controls, you can proceed to *Testing a Custom Control Using Dynamic Invoke* or to *Testing a Custom Control Using Automation Support*.

This procedure shows how a Flex application developer can create a spinner custom control in Flex. The spinner custom control that we create in this topic is used in several topics to illustrate the process of implementing and testing a custom control in Silk Test Classic.

The spinner custom control includes two buttons and a text box, as shown in the following graphic.



The user can click **Down** to decrement the value that is displayed in the text field and click **Up** to increment the value in the text field.

The custom control offers a public `CurrentValue` property that can be set and retrieved.

To define the custom control:

1. In the test application, define the layout of the control.

For example, for the spinner control type:

```
<?xml version="1.0" encoding="utf-8"?>
<customcontrols:SpinnerClass xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:controls="mx.controls.*" xmlns:customcontrols="customcontrols.*">
  <controls:Button id="downButton" label="Down" />
  <controls:TextInput id="text" enabled="false" />
  <controls:Button id="upButton" label="Up" />
</customcontrols:SpinnerClass>
```

2. Define the implementation of the custom control.

For example, for the spinner control type:

```
package
customcontrols
{
    import flash.events.MouseEvent;
    import mx.containers.HBox;
    import mx.controls.Button;
    import mx.controls.TextInput;
    import mx.core.UIComponent;
    import mx.events.FlexEvent;
    [Event(name="increment", type="customcontrols.SpinnerEvent")]
    [Event(name="decrement", type="customcontrols.SpinnerEvent")]

    public class SpinnerClass extends HBox
    {
        public var downButton : Button;
        public var upButton : Button;
        public var text : TextInput;
        public var ssss: SpinnerAutomationDelegate;
        private var _lowerBound : int = 0;
        private var _upperBound : int = 5;
        private var _value : int = 0;
        private var _stepSize : int = 1;

        public function SpinnerClass()
        {
            addEventListener(FlexEvent.CREATION_COMPLETE,
creationCompleteHandler);
        }

        private function creationCompleteHandler(event:FlexEvent) : void
        {
            downButton.addEventListener(MouseEvent.CLICK, downButtonClickHandler);
            upButton.addEventListener(MouseEvent.CLICK, upButtonClickHandler);
            updateText();
        }

        private function downButtonClickHandler(event : MouseEvent) : void
        {
            if(currentValue - stepSize >= lowerBound)
            {
                currentValue = currentValue - stepSize;
            }
            else
            {
                currentValue = upperBound - stepSize + currentValue - lowerBound +
1;
            }
            var spinnerEvent : SpinnerEvent = new
SpinnerEvent(SpinnerEvent.DECREMENT);
            spinnerEvent.steps = _stepSize;
            dispatchEvent(spinnerEvent);
        }

        private function upButtonClickHandler(event : MouseEvent) : void
        {
            if(currentValue <= upperBound - stepSize)
            {
                currentValue = currentValue + stepSize;
            }
            else
            {
                currentValue = lowerBound + currentValue + stepSize - upperBound -
```



```

1;
    }
    var spinnerEvent : SpinnerEvent = new
SpinnerEvent(SpinnerEvent.INCREMENT);
    spinnerEvent.steps = _stepSize;
    dispatchEvent(spinnerEvent);
    }

private function updateText() : void
{
    if(text != null)
    {
        text.text = _value.toString();
    }
}

public function get currentValue() : int
{
    return _value;
}

public function set currentValue(v : int) : void
{
    _value = v;
    if(v < lowerBound)
    {
        _value = lowerBound;
    }
    else if(v > upperBound)
    {
        _value = upperBound;
    }
    updateText();
}

public function get stepSize() : int
{
    return _stepSize;
}

public function set stepSize(v : int) : void
{
    _stepSize = v;
}

public function get lowerBound() : int
{
    return _lowerBound;
}

public function set lowerBound(v : int) : void
{
    _lowerBound = v;
    if(currentValue < lowerBound)
    {
        currentValue = lowerBound;
    }
}

public function get upperBound() : int
{
    return _upperBound;
}

```

```

public function set upperBound(v : int) : void
{
    _upperBound = v;
    if(currentValue > upperBound)
    {
        currentValue = upperBound;
    }
}
}
}

```

3. Define the events that the control uses.

For example, for the spinner control type:

```

package customcontrols
{
    import flash.events.Event;

    public class SpinnerEvent extends Event
    {
        public static const INCREMENT : String = "increment";
        public static const DECREMENT : String = "decrement";

        private var _steps : int;

        public function SpinnerEvent(eventName : String)
        {
            super(eventName);
        }

        public function set steps(value:int) : void
        {
            _steps = value;
        }

        public function get steps() : int
        {
            return _steps;
        }
    }
}

```

4. Proceed to *Implement Automation Support*.

Testing a Custom Control Using Dynamic Invoke

Silk Test Classic provides record and playback support for custom controls using dynamic invoke to interact with the custom control during replay. Use this low-effort approach when you want to access properties and methods of the custom control in the test application that Silk Test Classic does not expose. The developer of the custom control can also add methods and properties to the custom control specifically for making the control easier to test.

To test a custom control using dynamic invoke:

1. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.
2. Call dynamic methods on objects with the `DynamicInvoke` method.
3. Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method.
4. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList` method.
5. Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty` method.

Example

This example tests a spinner custom control that includes two buttons and a text box, as shown in the following graphic.



The user can click **Down** to decrement the value that is displayed in the text box and click **Up** to increment the value in the text box.

The custom control offers a public `CurrentValue` property that can be set and retrieved. The value in this example is 3.

To set the spinner's value to 4, type the following:

```
WINDOW spinner = Desktop.Find("//
FlexBox[@className=customcontrols.Spinner]")
spinner.SetProperty("CurrentValue", 4)
```

Testing a Custom Control Using Automation Support

Before you can test a custom control in Silk Test Classic, perform the following steps:

- Define the custom control in the test application.
- Implement automation support.

You can create specific automation support for the custom control. This additional automation support provides recording support and more powerful play-back support. To create automation support, the test application must be modified and the Open Agent must be extended.

After the test application has been modified and includes automation support, perform the following steps:

1. Open an existing Flex project or create a new project.
2. Click **File > New**.
The **New File** dialog box opens.
3. Choose **4Test include** and then click **OK**.
A new include file opens.
4. Type the custom control class information in the INC file and then click **Save**.

For example, the INC file for the `FlexSpinner` class looks like the following:

```
winclass FlexSpinner : FlexBox
tag "[FlexSpinner]"
builtin void Increment(INTEGER steps)
builtin void Decrement(INTEGER steps)
property stepSize
builtin INTEGER Get()
property lowerBound
builtin INTEGER Get()
property currentValue
builtin INTEGER Get()
builtin Set(INTEGER value)
property upperBound
builtin INTEGER Get()
```

5. Click **Options > Runtime Options** and in the **Use Files** field navigate to the custom control INC file.
6. Record and replay tests for the custom control.

Implementing Automation Support for a Custom Control

Before you can test a custom control, implement automation support, which is the automation delegate, in ActionScript for the custom control and compile that into the test application.

The following procedure uses a custom Flex spinner control to demonstrate how to implement automation support for a custom control. The spinner custom control includes two buttons and a text box, as shown in the following graphic.



The user can click **Down** to decrement the value that is displayed in the text box and click **Up** to increment the value in the text box.

The custom control offers a public `CurrentValue` property that can be set and retrieved.

1. Implement automation support, which is the automation delegate, in ActionScript for the custom control.

For further information about implementing an automation delegate, see the Adobe Live Documentation at http://livedocs.adobe.com/flex/3/html/help.html?content=functest_components2_14.html.

In this example, the automation delegate adds support for the methods `increment` and `decrement`. The example code for the automation delegate looks like this:

```
package customcontrols
{
    import flash.display.DisplayObject;
    import mx.automation.Automation;
    import customcontrols.SpinnerEvent;
    import mx.automation.delegates.containers.BoxAutomationImpl;
    import flash.events.Event;
    import mx.automation.IAutomationObjectHelper;
    import mx.events.FlexEvent;
    import flash.events.IEventDispatcher;
    import mx.preloaders.DownloadProgressbar;
    import flash.events.MouseEvent;
    import mx.core.EventPriority;

    [Mixin]
    public class SpinnerAutomationDelegate extends BoxAutomationImpl
    {
        public static function init(root:DisplayObject) : void
        {
            //register delegate for the automation
            Automation.registerDelegateClass(Spinner, SpinnerAutomationDelegate);
        }
        public function SpinnerAutomationDelegate(obj:Spinner)
        {
            super(obj);
            // listen to the events of interest (for recording)
            obj.addEventListener(SpinnerEvent.DECREMENT, decrementHandler);
            obj.addEventListener(SpinnerEvent.INCREMENT, incrementHandler);
        }

        protected function decrementHandler(event : SpinnerEvent) : void
        {
            recordAutomatableEvent(event);
        }

        protected function incrementHandler(event : SpinnerEvent) : void
        {
            recordAutomatableEvent(event);
        }
    }
}
```

```

protected function get spinner() : Spinner
{
    return uiComponent as Spinner;
}

//-----
// override functions
//-----

override public function get automationValue():Array
{
    return [ spinner.currentValue.toString() ];
}

private function replayClicks(button : IEventDispatcher, steps : int) :
Boolean
{
    var helper : IAutomationObjectHelper =
Automation.automationObjectHelper;
    var result : Boolean;
    for(var i:int; i < steps; i++)
    {
        helper.replayClick(button);
    }
    return result;
}

override public function replayAutomatableEvent(event:Event):Boolean
{
    if(event is SpinnerEvent)
    {
        var spinnerEvent : SpinnerEvent = event as SpinnerEvent;
        if(event.type == SpinnerEvent.INCREMENT)
        {
            return replayClicks(spinner.upButton, spinnerEvent.steps);
        }
        else if
        {
            return replayClicks(spinner.downButton, spinnerEvent.steps);
        }
        else
        {
            return false;
        }
    }
    else
    {
        return super.replayAutomatableEvent(event);
    }
}

// do not expose the child controls, which are the buttons and the
textfield, as individual controls
override public function get numAutomationChildren():int
{
    return 0;
}
}
}

```

2. To introduce the automation delegate to the Open Agent, create an XML file that describes the custom control.

The class definition file contains information about all instrumented Flex components. This file (or files) provides information about the components that can send events during recording and accept events for replay. The class definition file also includes the definitions for the supported properties.

The XML file for the spinner custom control looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<TypeInfoInformation>
  <ClassInfo Name="FlexSpinner" Extends="FlexBox">
    <Implementation Class="customcontrols.Spinner" />
    <Events>
      <Event Name="Decrement">
        <Implementation Class="customcontrols.SpinnerEvent"
          Type="decrement" />
        <Property Name="steps">
          <PropertyType Type="integer" />
        </Property>
      </Event>
    </Events>
    <Properties>
      <Property Name="lowerBound" accessType="read">
        <PropertyType Type="integer" />
      </Property>
      <Property Name="upperBound" accessType="read">
        <PropertyType Type="integer" />
      </Property>
      <!-- expose read and write access for the currentValue property -->
      <Property Name="currentValue" accessType="both">
        <PropertyType Type="integer" />
      </Property>
      <Property Name="stepSize" accessType="read">
        <PropertyType Type="integer" />
      </Property>
    </Properties>
  </ClassInfo>
</TypeInfoInformation>
```

3. Include the XML file for the custom control in the folder that includes all the XML files, which describe all classes, methods, and properties for the supported Flex controls.

Silk Test Classic contains several XML files that describe all classes, methods, and properties for the supported Flex controls. Those XML files are located in the `<silktest_install_directory>\ng\agent\plugins\com.borland.fastxd.techdomain.flex.agent_<version>\config\automationEnvironment` folder.

If you provide your own XML file, you must copy your XML file into this folder. When the Open Agent starts and initializes support for Flex, it reads the contents of this directory.

To test the Flex Spinner sample control, you must copy the `CustomControls.xml` file into this folder. If the Open Agent is currently running, restart it after you copy the file into the folder.

Now, you can test the custom control using Silk Test Classic.

Flex Class Definition File

The class definition file contains information about all instrumented Flex components. This file (or files) provides information about the components that can send events during recording and accept events for replay. The class definition file also includes the definitions for the supported properties.

Silk Test Classic contains several XML files that describe all classes, events, and properties for the common Flex common and specialized controls. Those XML files are located in the `<silktest_install_directory>\ng\agent\plugins\com.borland.fastxd.techdomain.flex.agent_<version>\config\automationEnvironment` folder.

If you provide your own XML file, you must copy your XML file into this folder. When the agent starts and initializes support for Apache Flex, it reads the contents of this directory.

The XML file has the following basic structure:

```
<TypeInfo>
<ClassInfo>
<Implementation />
<Events>
<Event />
...
</Events>
<Properties>
<Property />
...
</Properties>
</ClassInfo>
</TypeInfo>
```

Client/Server Application Support

Silk Test Classic provides built-in support for testing client/server applications including:

- .NET WinForms
- Java AWT applications
- Java SWT/RCP application
- Java Swing applications
- Windows-based applications

In a client/server environment, Silk Test Classic drives the client application by means of an Agent process running on each application's machine. The application then drives the server just as it always does. Silk Test Classic is also capable of driving the GUI belonging to a server or of directly driving a server database by running scripts that submit SQL statements to the database. These methods of directly manipulating the server application are intended to support testing in which the client application drives the server. For additional information on this capability, see *Testing Databases*.

Client/Server Testing Challenges

Silk Test Classic provides powerful support for testing client/server applications and databases in a networked environment. Testing multiple remote applications raises the level of complexity of QA engineering above that required for stand-alone application testing. Here are just a few of the testing methodology challenges raised by client/server testing:

- Managing simultaneous automatic regression tests on different configurations and platforms.
- Ensuring the reproducibility of client/server tests that modify a server database.
- Verifying the server operations of a client application independently, without relying on the application under test.
- Testing the concurrency features of a client/server application.
- Testing the intercommunication capabilities of networked applications.
- Closing down multiple failed applications and bringing them back to a particular base state (recovery control).
- Testing the functioning of the server application when driven at peak request rates and at maximum data rates (peak load and volume testing).
- Automated regression testing of multi-tier client/server architectures.

Verifying Tables in ClientServer Applications

This functionality is supported only if you are using the Classic Agent.

When verifying a table in a client/server application, that is, an object of the `Table` class or of a class derived from `Table`, you can verify the value of every cell in a specified range in the table using the **Table** tab in the **Verify Window** dialog box. For additional information on verifying tables in Web applications, see *Working with Borderless Tables*.

Specifying the range

You specify the range of cells to verify in the **Range** text boxes using the following syntax for the starting and ending cells in the range:

```
row_number : column_name
```

or

```
row_number : column_number
```

Example

Specifying the following in the **Range** text boxes of the **Verify Window** dialog box causes the value of every cell in rows 1 through 3 to be verified, starting with the column named ID and ending with the column named Company_Name:

From field: 1 : id

To field: 3 : company_name

After you specify a cell range in the **Verify Window** dialog box, you can click **Update** to display the values in the specified range.

Specifying a file to store the values

You specify a file to store the current values of the selected range in the **Table File Name** text box.

What happens

When you dismiss the **Verify Window** dialog box and paste the code into your script, the following occurs:

- The values that are currently in the table's specified cell range are stored in the file named in the **Table File Name** text box in the **Verify Window** dialog box.
- A `VerifyFileRangeValue` method is pasted in your script that references the file and the cell range you specified.

For example, the following `VerifyFileRangeValue` method call would be recorded for the preceding example:

```
table.VerifyFileRangeValue ("file.tbl", {{"1",  
"id"}, {"3", "company_name"}})
```

When you run your script, the values in the range specified in the second argument to `VerifyFileRangeValue` are compared to the values stored in the file referenced in the first argument to `VerifyFileRangeValue`.

For additional information, see the `VerifyFileRangeValue` method.

Evolving a Testing Strategy

There are several reasons for moving your QA program from local to remote testing:

- You may have a stand-alone application that runs on many different platforms and now you want to simultaneously drive testing on all the platforms from one Silk Test Classic host system.
- You may have been testing a client/server application as a single local application and now you want to drive multiple instances of the application so as to apply a heavier load to the server.

- You may want to upgrade your client/server testing so that your test cases can automatically initialize the server and recover from server failures— in addition to driving multiple application instances.
- You may need to test applications that have different user interfaces and that communicate as peers.

If you are already a Silk Test Classic user, you will find that your testing program can evolve in any of these directions while preserving large portions of your existing tests. This topic and related topics help you to evolve your testing strategy by showing the incremental steps you can take to move into remote testing.

Incremental Functional Test Design

Silk Test Classic simplifies and automates the classic QA testing methodology in which testing proceeds from the simplest cases to the most complex. This incremental functional testing methodology applies equally well in the client/ server environment, where testing scenarios typically proceed from the simplest functional testing of one instance of a client application, to functional and performance testing of a heavily loaded, multi-client configuration. Therefore, we recommend the following incremental progression for client/server testing:

- Perform functional testing on a single client application that is running on the same system as Silk Test Classic, with the server application on the same system (if possible).
- Perform functional testing on a single remote client application, with the server application on a separate system.
- Perform functional and concurrency testing on two remote client applications.
- Perform stress testing on a single client application running locally or remotely.
- Perform volume load testing on a configuration large enough to stress the server application.
- Perform peak load testing on a large configuration, up to the limits of the server, if possible.
- Perform performance testing on several sets of loads until you can predict performance.

Network Testing Types

Software testing can be categorized according to the various broad testing goals that are the focus of the individual tests. At a conceptual level, the kinds of automated application testing you can perform using Silk Test Classic in a networked environment are:

- Functional
- Configuration
- Concurrency

The ordering of this list conforms to the incremental functional testing methodology supported by Silk Test Classic. Each stage of testing depends for its effectiveness on the successful completion of the previous stage. Functional, configuration, and concurrency testing are variations of regression testing, which is a prerequisite for any type of load testing. You can use Silk Performer for load testing, stress testing, and performance testing.

You can perform functional testing with a single client machine. You can perform the first four types of test with a testbed containing only two clients. The last two testing types require a heavy multi-user load and so need a larger testbed.

Concurrency Testing

Concurrency testing tests two clients using the same server. This is a variation of functional testing that verifies that the server can properly handle simultaneous requests from two clients. The simplest form of concurrency testing verifies that two clients can make multiple non-conflicting server requests during the same period of time. This is a very basic sanity test for a client/server application.

To test for problems with concurrent access to the same database record, you need to write specific scripts that synchronize two clients to make requests of the same records in your server's databases at the same time. Your goal is to encounter faulty read/write locks, software deadlocks, or other concurrency problems.

Once the application passes the functional tests, you can test the boundary conditions that might be reached by large numbers of transactions.

Configuration Testing

A client/server application typically runs on multiple different platforms and utilizes a server that runs on one or more different platforms. A complete testing program needs to verify that every possible client platform can operate with every possible server platform. This implies the following combinations of tests:

- Test the client application and the server application when they are running on the same machine—if that is a valid operational mode for the application. This testing must be repeated for each platform that can execute in that mode.
- Test with the client and server on separate machines. This testing should be repeated for all different platform combinations of server and client.

Functional Testing

Before you test the multi-user aspects of a client/server application, you should verify the functional operation of a single instance of the application. This is the same kind of testing that you would do for a non-distributed application.

Once you have written scripts to test all the operations of the application as it runs on one platform, you can modify the scripts as needed for all other platforms on which the application runs. Testing multiple platforms thus becomes almost trivial. Moreover, many of the tests you script for functional testing can become the basis of your other types of testing. For example, you can easily modify the functional tests (or a subset of them) to use in load testing.

Peak Load Testing

Peak load testing is placing a load on the server for a short time to emulate the heaviest demand that would be generated at peak user times—for example, credit card verification between noon and 1 PM on Christmas Eve. This type of test requires a significant number of client systems. If you submit complex transactions to the server from each client in your test network, using minimal user setup, you can emulate the typical load of a much larger number of clients.

Your testbed may not have sufficient machines to place a heavy load on your server system — even if your clients are submitting requests at top speed. In this case it may be worthwhile to reconfigure your equipment so that your server is less powerful. An inadequate server configuration should enable you to test the server's management of peak server conditions.

Volume Testing

Volume testing is placing a heavy load on the server, with a high volume of data transfers, for 24 to 48 hours. One way to implement this is to use one set of clients to generate large amounts of new data and another set to verify the data, and to delete data to keep the size of the database at an appropriate level. In such a case, you need to synchronize the verification scripts to wait for the generation scripts. The 4Test script language makes this easy. Usually, you would need a very large test set to drive this type of server load, but if you under-configure your server you will be able to test the sections of the software that handle the outer limits of data capacity.

How 4Test Handles Script Deadlock

It is possible for a multi-threaded 4Test script to reach a state in which competing threads block one another, so that the script cannot continue. This is called a script deadlock. When the 4Test runtime environment detects a deadlock, it raises an exception and halts the deadlocked script.

Example

The following script will never exit successfully.

```
share INTEGER iIndex1 = 0
share INTEGER iIndex2 = 0

main ()
  parallel
    access iIndex1
    Sleep (1)
    access iIndex2
    Print ("Accessed iIndex1 and iIndex2")
  access iIndex2
  Sleep (1)
  access iIndex1
  Print ("Accessed iIndex2 and iIndex1")
```

Troubleshooting Configuration Test Failures

The test of your application may have failed for one of the reasons below. If the following suggestions do not address the problem, you can enable your extension manually.



Note: Unsupported and embedded browsers, other than AOL, are recognized as client/server applications.

The application may not have been ready to test

1. Click **Enable Extensions** on the **Basic workflow** bar.
2. On the **Enable Extensions** dialog box, select the application for which you want to enable extensions.
3. Close and restart your application. Make sure the application has finished loading, and then click **Test**.

Embedded browsers, other than AOL, are recognized as Client/Server applications

If you want to work with a web browser control embedded within an application, you must enable the extension manually.

Testing .NET Applications with the Open Agent

Silk Test Classic provides built-in support for testing .NET applications with the Open Agent.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Windows Forms Applications

Silk Test Classic provides built-in support for testing .NET Windows Forms (Win Forms) applications using the Open Agent as well as built-in support for testing .NET standalone and No-Touch Windows Forms (Win Forms) applications using the Classic Agent. However, side-by-side execution is supported only on standalone applications.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Locator Attributes for Windows Forms Applications

This functionality is supported only if you are using the Open Agent.

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

The attributes that Silk Test Classic supports for Windows Forms include:

- automationId
- caption. Supports wildcards ? and * .
- windowId
- priorLabel. For controls that do not have a caption, the priorLabel is used as the caption automatically. For controls with a caption, it may be easier to use the caption.



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Dynamically Invoking Windows Forms Methods

This functionality is supported only if you are using the Open Agent.

Dynamic invoke enables you to directly call methods, retrieve properties, or set properties, on an actual instance of a control in the application under test. You can also call methods and properties that are not available in the Silk Test Classic API for this control. Dynamic invoke is especially useful when you are working with custom controls, where the required functionality for interacting with the control is not exposed through the Silk Test Classic API.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList` method.

For example, to call a method named `SetTitle`, which requires the title to be set as an input parameter of type string, on an actual instance of a control in the application under test, type the following:

```
control.DynamicInvoke("SetTitle", {"my new title"})
```



Note: Typically, most properties are read-only and cannot be set.



Note: Reflection is used in most technology domains to call methods and retrieve properties.

The DynamicInvoke Method

For a Windows Forms or a WPF control, you can use the `DynamicInvoke` method to call the following methods:

- Public methods that the MSDN defines for the control.
- Public static methods that the MSDN defines.
- User-defined public static methods of any type.

First Example for the DynamicInvoke Method

For an object of the Silk Test Classic type `DataGrid`, you can call all methods that MSDN defines for the type `System.Windows.Forms.DataGrid`.

To call the method `IsExpanded` of the `System.Windows.Forms.DataGrid` class, use the following code:

```
//4Test code  
BOOLEAN isExpanded = (BOOLEAN)  
dataGrid.DynamicInvoke("IsExpanded", {3})
```

Second Example for the DynamicInvoke Method

To invoke the static method `String.Compare(String s1, String s2)` inside the AUT, use the following code:

```
//4Test code
INTEGER result =
mainWindow.DynamicInvoke("System.String.Compare", {"a", "b"});
```

The DynamicInvokeMethods Method

For a Windows Forms or a WPF control, you can use the `DynamicInvokeMethods` method to invoke a sequence of nested methods. You can call the following methods:

- Public methods that the MSDN defines for the control.
- Public static methods that the MSDN defines.
- User-defined public static methods of any type.

Example: Getting the Text Contents of a Cell in a Custom Data Grid

To get the text contents of a cell of a custom data grid from the Infragistics library, you can use the following C# code in the AUT:

```
string cellText =
dataGrid.Rows[rowIndex].Cells[columnIndex].Text;
```

The following C# code sample gets the text contents of the third cell in the first row:

```
string cellText = dataGrid.Rows[0].Cells[2];
```

Scripting the same example by using the `DynamicInvokeMethods` method generates a relatively complex script, because you have to pass five methods with their corresponding parameters to the `DynamicInvokeMethods` method:

```
INTEGER rowIndex = 0
INTEGER columnIndex = 2

LIST OF STRING names = { ... }
  "Rows"           // Get the list of rows from the grid.
  "get_Item"       // Get a specific row from the list of rows by
                    // using the indexer method.
  "Cells"          // Get the list of cells from the the row.
  "get_Item"       // Get a specific cell from the list of cells
                    // by using the indexer method.
  "Text"           // Get the text of the cell.

LIST OF LIST parameters = { ... }
  {}               // Parameters for the Rows property.
  {rowIndex}       // Parameters for the get_Item method.
  {}               // Parameters for the Cells property.
  {columnIndex}    // Parameters for the get_Item method.
  {}               // Parameters for the Text property.

dataGrid.DynamicInvokeMethods(names, parameters)
```

Supported Methods and Properties

The following methods and properties can be called:

- Methods and properties that Silk Test Classic supports for the control.
- All public methods and properties that the MSDN defines for the control.

- If the control is a custom control that is derived from a standard control, all methods and properties from the standard control can be called.

Supported Parameter Types

The following parameter types are supported:

- All built-in Silk Test Classic types

Silk Test Classic types includes primitive types (such as boolean, int, string), lists, and other types (such as Point and Rect).

- Enum types

Enum parameters must be passed as string. The string must match the name of an enum value. For example, if the method expects a parameter of the .NET enum type `System.Windows.Visibility` you can use the string values of `Visible`, `Hidden`, or `Collapsed`.

- .NET structs and objects

.NET struct and object parameters must be passed as a list. The elements in the list must match one constructor for the .NET object in the test application. For example, if the method expects a parameter of the .NET type `System.Windows.Vector`, you can pass a list with two integers. This works because the `System.Windows.Vector` type has a constructor with two integer arguments.

Returned Values

The following values are returned for properties and methods that have a return value:

- The correct value for all built-in Silk Test Classic types. These types are listed in the *Supported Parameter Types* section.
- All methods that have no return value return `NULL`.

Suppressing Controls (Open Agent)

This functionality is supported only if you are using the Open Agent.

To simplify the object hierarchy and to shorten the length of the lines of code in your test scripts and functions, you can suppress the controls for certain unnecessary classes in the following technologies:

- Win32.
- Java AWT/Swing.
- Java SWT/Eclipse.
- Windows Presentation Foundation (WPF).

For example, you might want to ignore container classes to streamline your test cases.

To suppress specific controls:

1. Click **Options > Recorder**. The **Recording Options** dialog box opens.
2. Click the **Transparent Classes** tab.
3. Type the name of the class that you want to ignore during recording and playback into the text box.

If the text box already contains classes, add the new classes to the end of the list. Separate the classes with a comma. For example, to ignore both the `AOL.Toolbar` and the `_AOL.Toolbar` class, type `AOL.Toolbar, _AOL.Toolbar` into the text box.

The `OPT_TRANSPARENT_CLASSES` option is set to true for these classes.

4. Click **OK**. The `OPT_TRANSPARENT_CLASSES` option is set to true for these classes, which means the classes are added to the list of the classes that are ignored during recording and playback.

Determining the priorLabel in the Windows Forms Technology Domain

To determine the priorLabel in the Windows Forms technology domain, all labels and groups in the same window as the target control are considered. The decision is then made based upon the following criteria:

- Only labels either above or to the left of the control, and groups surrounding the control, are considered as candidates for a priorLabel.
- In the simplest case, the label closest to the control is used as the priorLabel.
- If two labels have the same distance to the control, the priorLabel is determined based upon the following criteria:
 - If one label is to the left and the other above the control, the left one is preferred.
 - If both labels are to the left of the control, the upper one is preferred.
 - If both labels are above the control, the left one is preferred.
- If the closest control is a group control, first all labels within the group are considered according to the rules specified above. If no labels within the group are eligible, then the caption of the group is used as the priorLabel.

WPF Applications

This functionality is supported only if you are using the Open Agent.

Silk Test Classic provides built-in support for testing Windows Presentation Foundation (WPF) applications using the Open Agent. Silk Test Classic supports standalone WPF applications and can record and play back controls in .NET version 3.5 or later.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

WPF applications support hierarchical object recognition and dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. You can use both recognition methods within a single test case if necessary. Use the method best suited to meet your test requirements.


When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Existing test cases that use dynamic object recognition without locator keywords in an INC file will continue to be supported. You can replay these tests, but you cannot record new tests with dynamic object recognition without locator keywords in an INC file.

When you create a new WPF project, Silk Test Classic uses the Open Agent by default.

Supported Controls for WPF

Silk Test Classic includes record and replay support for WPF controls. In Silk Test 2009, WPF replay support was provided. However, with the release of Silk Test 2010, the earlier WPF controls, which were prefixed with MSUIA, are deprecated and users should use the new WPF technology domain instead. When you record new test cases, Silk Test Classic automatically uses the new WPF technology domain.

 **Note:** If you have an existing project that includes scripts that use the earlier MSUIA technology domain, the test cases will no longer work.

For a complete list of the controls available for WPF testing, see the *WPF Class Reference*.

Locator Attributes for Windows Presentation Foundation (WPF) Controls

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Silk Test Classic supports the following locator attributes for WPF controls:

- *automationId*
- *caption*
- *className*
- *name*



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Dynamic Object Recognition

To identify components within WPF scripts, you can specify the *automationId*, *caption*, *className*, or *name*. The name that is given to an element in the application is used as the *automationId* attribute for the locator if available. As a result, most objects can be uniquely identified using only this attribute. For example, a locator with an *automationId* might look like: //

```
WPFButton[@automationId='okButton']"
```

If you define an *automationId* and any other attribute, only the *automationId* is used during replay. If there is no *automationId* defined, the *name* is used to resolve the component. If neither a *name* nor an *automationId* are defined, the *caption* value is used. If no caption is defined, the *className* is used. We recommend using the *automationId* because it is the most useful property.

Attribute Type	Description	Example
<i>automationId</i>	An ID that was provided by the developer of the test application.	//WPFButton[@automationId='okButton']"
<i>name</i>	The name of a control. The Visual Studio designer automatically assigns a name to every control that is created with the designer. The application developer uses this name to identify the control in the application code.	//WPFButton[@name='okButton']"
<i>caption</i>	The text that the control displays. When testing a localized application in multiple languages, use the <i>automationId</i> or <i>name</i> attribute instead of the <i>caption</i> .	//WPFButton[@automationId='Ok']"
<i>className</i>	The simple .NET class name (without namespace) of the WPF control. Using the class name attribute can help to identify a custom control that is derived from a standard WPF control that Silk Test Classic recognizes.	//WPFButton[@className='MyCustomButton']"

During recording, Silk Test Classic creates a locator for a WPF control by using the *automationId*, *name*, *caption*, or *className* attributes in the order that they are listed in the preceding table. For example, if a control has a *automationId* and a *name*, Silk Test Classic uses the *automationId* when creating the locator.

The following example shows how an application developer can define a *name* and an *automationId* for a WPF button in the XAML code of the application:

```
<Button Name="okButton" AutomationProperties.AutomationId="okButton"
Click="okButton_Click">Ok</Button>
```

Classes that Derive from the WPFItemsControl Class

Silk Test Classic can interact with classes that derive from `WPFItemsControl`, such as `WPFListBox`, `WPFTreeView`, and `WPFMenu`, in two ways:

Working with the control Most controls contain methods and properties for typical use cases. The items are identified by text or index.

For example:

```
listBox.Select("Banana")
listBox.Select(2)
tree.Expand("/Fruit/Banana")
```

Working with individual items For example `WPFListBoxItem`, `WPFTreeViewItem`, or `WPFMenuItem`. For advanced use cases, use individual items. For example, use individual items for opening the context menu on a specific item in a list box, or clicking a certain position relative to an item.

Custom WPF Controls

Generally, Silk Test Classic provides record and playback support for all standard WPF controls.

Silk Test Classic handles custom controls based on the way the custom control is implemented. You can implement custom controls by using the following approaches:

Deriving classes from UserControl This is a typical way to create compound controls. Silk Test Classic recognizes these user controls as `WPFUserControl` and provides full support for the contained controls.

Deriving classes from standard WPF controls, such as ListBox Silk Test Classic treats these controls as an instance of the standard WPF control that they derive from. Record, playback, and recognition of children may not work if the user control behavior differs significantly from its base class implementation.

Using standard controls that use templates to change their visual appearance Low-level replay might not work in certain cases. Switch to high-level replay in such cases.

Silk Test Classic filters out certain controls that are typically not relevant for functional testing. For example, controls that are used for layout purposes are not included. However, if a custom control derives from an excluded class, specify the name of the related WPF class to expose the filtered controls during recording and playback.

Setting WPF Classes to Expose During Recording and Playback

Silk Test Classic filters out certain controls that are typically not relevant for functional testing. For example, controls that are used for layout purposes are not included. However, if a custom control derives from an excluded class, specify the name of the related WPF class to expose the filtered controls during recording and playback.

Specify the names of any WPF classes that you want to expose during recording and playback. For example, if a custom class called `MyGrid` derives from the WPF `Grid` class, the objects of the `MyGrid` custom class are not available for recording and playback. `Grid` objects are not available for recording and playback because the `Grid` class is not relevant for functional testing since it exists only for layout

purposes. As a result, `Grid` objects are not exposed by default. In order to use custom classes that are based on classes that are not relevant to functional testing, add the custom class, in this case `MyGrid`, to the **OPT_WPF_CUSTOM_CLASSES** option. Then you can record, playback, find, verify properties, and perform any other supported actions for the specified classes.

1. Click **Options**.
2. Click **Recorder**.
3. In the **Custom WPF class names** grid, type the name of the class that you want to expose during recording and playback.
Separate class names with a comma.
4. Click **OK**.

Dynamically Invoking WPF Methods

Dynamic invoke enables you to directly call methods, retrieve properties, or set properties, on an actual instance of a control in the application under test. You can also call methods and properties that are not available in the Silk Test Classic API for this control. Dynamic invoke is especially useful when you are working with custom controls, where the required functionality for interacting with the control is not exposed through the Silk Test Classic API.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList` method.

For example, to call a method named `SetTitle`, which requires the title to be set as an input parameter of type string, on an actual instance of a control in the application under test, type the following:

```
control.DynamicInvoke("SetTitle", {"my new title"})
```



Note: Typically, most properties are read-only and cannot be set.



Note: Reflection is used in most technology domains to call methods and retrieve properties.

The DynamicInvoke Method

For a Windows Forms or a WPF control, you can use the `DynamicInvoke` method to call the following methods:

- Public methods that the MSDN defines for the control.
- Public static methods that the MSDN defines.
- User-defined public static methods of any type.

First Example for the DynamicInvoke Method

For an object of the Silk Test Classic type `DataGrid`, you can call all methods that MSDN defines for the type `System.Windows.Forms.DataGrid`.

To call the method `IsExpanded` of the `System.Windows.Forms.DataGrid` class, use the following code:

```
//4Test code  
BOOLEAN isExpanded = (BOOLEAN)  
dataGrid.DynamicInvoke("IsExpanded", {3})
```

Second Example for the DynamicInvoke Method

To invoke the static method `String.Compare(String s1, String s2)` inside the AUT, use the following code:

```
//4Test code
INTEGER result =
mainWindow.DynamicInvoke("System.String.Compare", {"a", "b"});
```

The DynamicInvokeMethods Method

For a Windows Forms or a WPF control, you can use the `DynamicInvokeMethods` method to invoke a sequence of nested methods. You can call the following methods:

- Public methods that the MSDN defines for the control.
- Public static methods that the MSDN defines.
- User-defined public static methods of any type.

Example: Getting the Text Contents of a Cell in a Custom Data Grid

To get the text contents of a cell of a custom data grid from the Infragistics library, you can use the following C# code in the AUT:

```
string cellText =
dataGrid.Rows[rowIndex].Cells[columnIndex].Text;
```

The following C# code sample gets the text contents of the third cell in the first row:

```
string cellText = dataGrid.Rows[0].Cells[2];
```

Scripting the same example by using the `DynamicInvokeMethods` method generates a relatively complex script, because you have to pass five methods with their corresponding parameters to the `DynamicInvokeMethods` method:

```
INTEGER rowIndex = 0
INTEGER columnIndex = 2

LIST OF STRING names = { ... }
"Rows"           // Get the list of rows from the grid.
"get_Item"       // Get a specific row from the list of rows by
using the indexer method.
"Cells"          // Get the list of cells from the the row.
"get_Item"       // Get a specific cell from the list of cells
by using the indexer method.
"Text"           // Get the text of the cell.

LIST OF LIST parameters = { ... }
{}               // Parameters for the Rows property.
{rowIndex}      // Parameters for the get_Item method.
{}               // Parameters for the Cells property.
{columnIndex}  // Parameters for the get_Item method.
{}               // Parameters for the Text property.

dataGrid.DynamicInvokeMethods(names, parameters)
```

Supported Methods and Properties

The following methods and properties can be called:

- Methods and properties that Silk Test Classic supports for the control.
- All public methods and properties that the MSDN defines for the control.

- If the control is a custom control that is derived from a standard control, all methods and properties from the standard control can be called.

Supported Parameter Types

The following parameter types are supported:

- All built-in Silk Test Classic types

Silk Test Classic types includes primitive types (such as boolean, int, string), lists, and other types (such as Point and Rect).
- Enum types

Enum parameters must be passed as string. The string must match the name of an enum value. For example, if the method expects a parameter of the .NET enum type `System.Windows.Visibility` you can use the string values of `Visible`, `Hidden`, or `Collapsed`.
- .NET structs and objects

.NET struct and object parameters must be passed as a list. The elements in the list must match one constructor for the .NET object in the test application. For example, if the method expects a parameter of the .NET type `System.Windows.Vector`, you can pass a list with two integers. This works because the `System.Windows.Vector` type has a constructor with two integer arguments.

Returned Values

The following values are returned for properties and methods that have a return value:

- The correct value for all built-in Silk Test Classic types. These types are listed in the *Supported Parameter Types* section.
- All methods that have no return value return `NULL`.
- A string for all other types

Call `ToString` on returned .NET objects to retrieve the string representation

Example

A custom calculator control has a `Reset` method and an `Add` method, which performs an addition of two numbers. You can use the following code to call the methods directly from your tests:

```
customControl.DynamicInvoke("Reset")
REAL sum = customControl.DynamicInvoke("Add", {1,2})
```

The calculator control also has a `LastCalculationResult` property. You can use the following code to read the property:

```
REAL lastResult =
customControl.GetProperty("LastCalculationResult")
```

Setting Pre-Fill During Recording and Replaying

This functionality is supported only if you are using the Open Agent.

Defines whether items in a `WPFItemsControl`, like `WPFComboBox` or `WPFListBox`, are pre-filled during recording and playback. WPF itself lazily loads items for certain controls, so these items are not available for Silk Test Classic if they are not scrolled into view. Turn pre-filling on, which is the default setting, to additionally access items that are not accessible without scrolling them into view. However, some applications have problems when the items are pre-filled by Silk Test Classic in the background, and these applications can therefore crash. In this case turn pre-filling off.

1. Click **Options**.

2. Click **Recorder**.
3. In the **Pre-fill items** area, check the **OPT_WPF_PREFILL_ITEMS** check box.
4. Click **OK**.

WPF Class Reference

When you configure a WPF application, Silk Test Classic automatically provides built-in support for testing standard WPF controls.

Microsoft Silverlight Applications

Microsoft Silverlight (Silverlight) is an application framework for writing and running rich internet applications, with features and purposes similar to those of Adobe Flash. The run-time environment for Silverlight is available as a plug-in for most web browsers.

Silk Test Classic provides built-in support for testing Silverlight applications with the Open Agent. Silk Test Classic supports Silverlight applications that run in a browser as well as out-of-browser and can record and play back controls in Silverlight.

The following applications, that are based on Silverlight, are supported:

- Silverlight applications that run in Internet Explorer.
- Silverlight applications that run in Mozilla Firefox.
- Out-of-Browser Silverlight applications.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Silverlight applications support dynamic object recognition. You can create tests for dynamic object recognition in your test environment.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Silk Test Classic includes record and replay support for Silverlight controls. For a complete list of the controls available for Silverlight testing, see the *Silverlight Class Reference*.

The support for testing Silverlight applications in Microsoft Windows XP requires the installation of Service Pack 3 and the Update for Windows XP with the Microsoft User Interface Automation that is provided in Windows 7. You can download the update from <http://www.microsoft.com/download/en/details.aspx?id=13821>.



Note: The Microsoft User Interface Automation needs to be installed for the Silverlight support. If you are using a Windows operating system and the Silverlight support does not work, you can install the update with the Microsoft User Interface Automation, which is appropriate for your operating system, from <http://support.microsoft.com/kb/971513>.

Locator Attributes for Silverlight Controls

Silk Test Classic supports the following locator attributes for Silverlight controls:

- *automationId*
- *caption*
- *className*
- *name*
- All dynamic locator attributes




Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Dynamic Object Recognition

To identify components within Silverlight scripts, you can specify the *automationId*, *caption*, *className*, *name* or any dynamic locator attribute. The *automationId* can be set by the application developer. For example, a locator with an *automationId* might look like `//SLButton[@automationId="okButton"]`.

We recommend using the *automationId* because it is typically the most useful and stable attribute.

Attribute Type	Description	Example
automationId	An identifier that is provided by the developer of the application under test. The Visual Studio designer automatically assigns an <i>automationId</i> to every control that is created with the designer. The application developer uses this ID to identify the control in the application code.	<code>// SLButton[@automationId="okButton"]</code>
caption	The text that the control displays. When testing a localized application in multiple languages, use the <i>automationId</i> or <i>name</i> attribute instead of the <i>caption</i> .	<code>//SLButton[@caption="OK"]</code>
className	The simple .NET class name (without namespace) of the Silverlight control. Using the <i>className</i> attribute can help to identify a custom control that is derived from a standard Silverlight control that Silk Test recognizes.	<code>// SLButton[@className='MyCustomButton']</code>
name	The name of a control. Can be provided by the developer of the application under test.	<code>//SLButton[@name="okButton"]</code>

 **Attention:** The name attribute in XAML code maps to the locator attribute *automationId*, not to the locator attribute name.

During recording, Silk Test Classic creates a locator for a Silverlight control by using the *automationId*, *name*, *caption*, or *className* attributes in the order that they are listed in the preceding table. For example, if a control has a *automationId* and a *name*, Silk Test Classic uses the *automationId* when creating the locator.

The following table shows how an application developer can define a Silverlight button with the text `Ok` in the XAML code of the application:

XAML Code for the Object	Locator to Find the Object from Silk Test
<code><Button>Ok</Button></code>	<code>//SLButton[@caption="OK"]</code>
<code><Button Name="okButton">Ok</Button></code>	<code>//SLButton[@automationId="okButton"]</code>
<code><Button AutomationProperties.AutomationId="okButton">Ok</Button></code>	<code>//SLButton[@automationId="okButton"]</code>
<code><Button AutomationProperties.Name="okButton">Ok</Button></code>	<code>//SLButton[@name="okButton"]</code>

Dynamically Invoking Silverlight Methods

You can call methods, retrieve properties, and set properties on controls that Silk Test Classic does not expose by using the dynamic invoke feature. This feature is useful for working with custom controls and for working with controls that Silk Test Classic supports without customization.

Call dynamic methods on Silverlight objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a Silverlight control, use the `GetDynamicMethodList()` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a Silverlight control, use the `GetDynamicMethodList()` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty()` method. To retrieve a list of supported dynamic properties for a Silverlight control, use the `GetPropertyList()` method.



Note: Typically, most properties are read-only and cannot be set.

Supported Parameter Types

All built-in Silk Test Classic types Silk Test Classic types include primitive types, for example boolean, int, and string, lists, and other types, for example Point and Rect.

Enum types Enum parameters must be passed as string. The string must match the name of an enum value. For example, if the method expects a parameter of the .NET enum type `System.Windows.Visibility` you can use the string values of `Visible`, `Hidden`, or `Collapsed`.

.NET structs and objects Pass .NET struct and object parameters as a list. The elements in the list must match one constructor for the .NET object in the test application. For example, if the method expects a parameter of the .NET type `System.Windows.Vector`, you can pass a list with two integers. This works because the `System.Windows.Vector` type has a constructor with two integer arguments.

Other controls Control parameters can be passed as `TestObject`.

Returned Values

The following values are returned for properties and methods that have a return value:

- The correct value for all built-in Silk Test Classic types.
- All methods that have no return value return NULL.
- A string for all other types.

To retrieve this string representation, call the `ToString()` method on returned .NET objects in the application under test.

Example

A `TabItem` in Silverlight, which is an item in a `TabControl`.

```
tabItem.DynamicInvoke("SelectedItemPattern.Select")
mySilverlightObject.GetProperty("IsPassword")
```

Scrolling in Silverlight

Silk Test Classic provides two different sets of scrolling-related methods and properties, depending on the Silverlight control.

- The first type of controls includes controls that can scroll by themselves and therefore do not expose the scrollbars explicitly as children. For example combo boxes, panes, list boxes, tree controls, data grids, auto complete boxes, and others.
- The second type of controls includes controls that cannot scroll by themselves but expose scrollbars as children for scrolling. For example text fields.

This distinction in Silk Test Classic exists because the controls in Silk Test Classic implement scrolling in those two ways.

Controls that support scrolling

In this case, scrolling-related methods and property are available for the control that contains the scrollbars. Therefore, Silk Test Classic does not expose scrollbar objects.

Examples

The following command scrolls a list box to the bottom:

```
listBox.SetVerticalScrollPercent(100)
```

The following command scrolls the list box down by one unit:

```
listBox.ScrollVertical(ScrollAmount.SmallIncrement)
```

Controls that do not support scrolling

In this case the scrollbars are exposed. No scrolling-related methods and properties are available for the control itself. The horizontal and vertical scrollbar objects enable you to scroll in the control by specifying the increment or decrement, or the final position, as a parameter in the corresponding API functions. The increment or decrement can take the values of the ScrollAmount enumeration. For additional information, refer to the Silverlight documentation. The final position is related to the position of the object, which is defined by the application designer.

Examples

The following command scrolls a vertical scrollbar within a text box to position 15:

```
textBox.SLVerticalScrollBar().ScrollToPosition(15)
```

The following command scrolls a vertical scrollbar within a text box to the bottom:

```
textBox.SLVerticalScrollBar().ScrollToMaximum()
```

Troubleshooting when Testing Silverlight Applications

Silk Test Classic cannot see inside the Silverlight application and no green rectangles are drawn during recording

The following reasons may cause Silk Test Classic to be unable to see inside the Silverlight application:

Reason	Solution
You use a Mozilla Firefox version prior to 4.0.	Use Mozilla Firefox 4.0 or later.
You use a Silverlight version prior to 3.	Use Silverlight 3 (Silverlight Runtime 4) or Silverlight 4 (Silverlight Runtime 4).
Your Silverlight application is running in windowless mode.	<p>Silk Test Classic does not support Silverlight applications that run in windowless mode. To test such an application, you need to change the Web site where your Silverlight application is running. Therefore you need to set the <code>windowless</code> parameter in the object tag of the HTML or ASPX file, in which the Silverlight application is hosted, to <code>false</code>.</p> <p>The following sample code sets the <code>windowless</code> parameter to <code>false</code>:</p> <pre><object ... > <param name="windowless"</pre>

Reason	Solution
	<pre>value="false" /> ... </object></pre>

Silverlight Class Reference

When you configure a Silverlight application, Silk Test Classic automatically provides built-in support for testing standard Silverlight controls.

Testing Java AWT/Swing Applications with the Open Agent

Silk Test Classic provides built-in for testing stand-alone Java applications developed using supported Java virtual machines and for testing Java applets using supported browsers. You must configure Silk Test Classic Java support before using it. When you configure a Java AWT/Swing application or applet, Silk Test Classic automatically provides support for testing standard Java AWT/Swing controls. You can also test Java SWT controls embedded in Java AWT/Swing applications or applets as well as Java AWT/Swing controls embedded in Java SWT applications.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Testing Standard Java Objects and Custom Controls

Any single Java application or applet may contain a mix of standard and custom Java objects. With Java support, you can test both types of visible Java objects in applications and applets that you develop using the Java Development Kit (JDK)/Java Runtime Environment (JRE).

Standard Java objects are often defined in class libraries. The Java support of Silk Test Classic lets you record and play back tests against standard controls by providing 4Test definitions for many Java classes defined in the following class libraries:

- Abstract Windowing Toolkit (AWT)
- Java Foundation Class (JFC), which includes the Swing set of GUI components
- Standard Widget Toolkit (SWT)
- Symantec Visual Café Itools (only for the Classic Agent)

If you are using the Classic Agent, you can use the `setName("<desiredwindow ID>")` method to create a window ID that Silk Test Classic will detect. `setName()` is a method inherited from class `java.awt.Component`, so it should work for most, if not all, of the Java classes that Silk Test Classic can detect. If you are using the Open Agent, the equivalent of the `setName` method is the `Name` property of the `AWTComponent` class.

By contrast, custom controls often use native properties and native methods written in Java. Increasingly, custom controls also take the form of JavaBeans, which are reusable platform-independent software components written in Java. Developers frequently design custom controls to achieve functionality that is not available in standard class libraries. You can test custom Java objects, including JavaBeans, using the Silk Test Classic Java support.

The Silk Test Classic approach to testing custom Java objects is to give you direct access to their native methods and properties. A major advantage of this methodology is that it obviates the need to write your own native methods.

The procedure for testing custom Java objects is simple: Record a class for the custom control, then save the class definition in an include file. The class definition includes the native methods you can call and native properties you can verify from your 4Test script.

The predefined property sets supplied with Silk Test Classic have not been customized for Java; however, you can modify these property sets. For additional information about editing existing property sets or creating new property sets, see *Creating a Property Set*.

Recording and Playing Back JFC Menus

For Sun JDK v1.4 or later, Silk Test Classic can record and play back regular menus that conform to the Windows standard, as well as JFC heavyweight and lightweight pop-up menus.

Recording and Playing Back Java AWT Menus

Unlike JFC menus, AWT menus are not conform to the Java component-container paradigm. Therefore, their behavior is different than that of the JFC menus, and is independent of the JVM version. Silk Test Classic can record regular AWT menus for all versions of the JDK.

For context menus that are conform to the Windows standard, which means that they can be opened with a right-click, Silk Test Classic can play back, but not record, the context menus for all versions of the JDK.

For AWT popup menus that are not conform to the Windows standard, Silk Test Classic cannot record or play back for all versions of the JDK. The `JavaAwtPopupMenu` class is available for playback only. Silk Test Classic is not able to record it and you must hand script any interaction with such a menu.

Object Recognition for Java AWT/Swing Applications

Java AWT/Swing applications support hierarchical object recognition and dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Using custom class attributes becomes even more powerful when it is used in combination with dynamic object recognition.

To test Java AWT/Swing applications using hierarchical object recognition, record a test for the application that you want to test. Then, replay the tests at your convenience.

Agent Support for Java AWT/Swing Applications

You can test Java AWT/Swing applications using the Classic Agent or the Open Agent. When you create a new Java AWT/Swing project, Silk Test Classic uses the Open Agent by default. However, you can use both the Open Agent and the Classic Agent within a single Java AWT/Swing environment. Certain functions and methods run on the Classic Agent only. As a result, if you are running an Open Agent project, the Classic Agent may also open because a function or method requires the Classic Agent.

When you are using the Classic agent to test Java AWT/Swing applications, Silk Test Classic uses the Sun JDK by default.

Supported Controls for Java AWT/Swing Applications

For a complete list of the record and replay controls available for Java AWT/Swing testing with the Open Agent, refer to the *Java AWT and Swing Class Reference* in the *4Test Language* section of the Help.

For a complete list of the record and replay controls available for Java AWT/Swing testing with the Classic Agent, refer to the *Java AWT Classes for the Classic Agent* and the *Java JFC Classes* in the *4Test Language* section of the Help.

Java AWT and Swing Class Reference

When you configure a Java AWT/Swing application, Silk Test Classic automatically provides built-in support for testing standard Java AWT/Swing controls.

Configuring a Test Application that Uses the Java Network Launching Protocol (JNLP)

This functionality is supported only if you are using the Open Agent.

Applications that start using the Java Network Launching Protocol (JNLP) require additional configuration in Silk Test Classic. Because these applications are started from the Web, you must manually configure the application configuration to start the actual application as well as the application that launches the "Web Start". Otherwise, the test will fail on playback unless the application is already running.

1. Record a test case for the application that you want to test.
2. In the INC file, replace the `const sCmdLine = value` with the command line pattern that includes the absolute path to `javaws.exe` and the URL to the Web Start.

For example, to use the SwingSet3 JNLP application, type `const sCmdLine = "%ProgramFiles%\Java\jre6\bin\javaws.exe http://download.java.net/javadesktop/swingset3/SwingSet3.jnlp"`

When you replay the test case, the JNLP application starts as expected.

Custom Attributes

This functionality is supported only if you are using the Open Agent.

Add custom attributes to a test application to make a test more stable. You can use custom attributes with the following technologies:

- Java SWT
- Swing
- WPF
- xBrowser
- Windows Forms
- SAP

For example, in Java SWT, the developer implementing the GUI can define an attribute (for example, `silkTestAutomationId`) for a widget that uniquely identifies the widget in the application. A tester using Silk Test Classic can then add that attribute to the list of custom attributes (in this case, `silkTestAutomationId`), and can identify controls by that unique ID. Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index will change whenever another widget is added before the one you have defined already.

If more than one object is assigned the same custom attribute value, all the objects with that value will return when you call the custom attribute. For example, if you assign the unique ID, 'loginName' to two different fields, both fields will return when you call the `loginName` attribute.

First, enable custom attributes for your application and then create the test.

Recording tests that use dynamic object recognition

Using custom class attributes becomes even more powerful when it is used in combination with dynamic object recognition. For example, if you create a button in the application that you want to test using the following code:

```
Button myButton = Button(parent, SWT.NONE);
myButton.setData("SilkTestAutomationId", "myButtonId");
```

To add the attribute to your XPath query string in your test case, you can use the following query:

```
Window button = Desktop.Find("//
PushButton[@SilkTestAutomationId='myButton']")
```


Locator Attributes for Java AWT/Swing Controls

This functionality is supported only if you are using the Open Agent.

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Silk Test Classic supports the following locator attributes for Java AWT/Swing controls:

- caption
- name
- accessibleName
- priorlabel (For controls that do not have a caption, the priorlabel is used as the caption automatically. For controls with a caption, it may be easier to use the caption.)
- Swing only: All custom object definition attributes set in the widget with `SetClientProperty("propertyName", "propertyValue")`.

 **Note:** Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Dynamically Invoking Java Methods

This functionality is supported only if you are using the Open Agent.

You can call methods, retrieve properties, and set properties on controls that Silk Test Classic does not expose by using the dynamic invoke feature. This feature is useful for working with custom controls and for working with controls that Silk Test Classic supports without customization.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList()` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty()` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList()` method.

 **Note:** Typically, most properties are read-only and cannot be set.

Supported Methods and Properties

The following methods and properties can be called:

- Methods and properties that Silk Test Classic supports for the control.
- All public methods of the SWT, AWT, or Swing widget.

- If the control is a custom control that is derived from a standard control, all methods and properties from the standard control can be called.

Supported Parameter Types

The following parameter types are supported:

All built-in Silk Test Classic types. Silk Test Classic types includes primitive types (such as boolean, int, string), lists, and other types (such as Point and Rect).

Enum types. Enum parameters must be passed as string. The string must match the name of an enum value. For example, if the method expects a parameter of the enum type, `java.sql.ClientInfoStatus` you can use the string values of `REASON_UNKNOWN`, `REASON_UNKNOWN_PROPERTY`, `REASON_VALUE_INVALID`, or `REASON_VALUE_TRUNCATED`.

Example

A custom calculator control has a `Reset` method and an `Add` method, which performs an addition of two numbers. You can use the following code to call the methods directly from your tests:

```
customControl.Invoke("Reset")
REAL sum = customControl.DynamicInvoke("Add", {1,2})
```

Determining the priorLabel in the Java AWT/Swing Technology Domain

This functionality is supported only if you are using the Open Agent.

To determine the `priorLabel` in the Java AWT/Swing technology domain, all labels and groups in the same window as the target control are considered. The decision is then made based upon the following criteria:

- Only labels either above or to the left of the control, and groups surrounding the control, are considered as candidates for a `priorLabel`.
- If a parent of the control is a `JViewport` or a `ScrollPane`, the algorithm works as if the parent is the window that contains the control, and nothing outside is considered relevant.
- In the simplest case, the label closest to the control is used as the `priorLabel`.
- If two labels have the same distance to the control, and one is to the left and the other above the control, the left one is preferred.
- If no label is eligible, the caption of the closest group is used.

Supported Browsers for Testing Java Applets

Silk Test Classic supports the following browsers for testing Java applets:

- For the Classic Agent: Internet Explorer 7 using the Java plug-in.
- For the Open Agent: All supported versions of Internet Explorer and Mozilla Firefox.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Overview of JavaScript Support

Silk Test Classic provides support for executing JavaScript code within a Web application. You can test applications that include JavaScript by performing one of the following tasks:

- Configuring an xBrowser application that uses the Open Agent.
- Enabling extensions for a generic application that uses the Classic Agent.

The type of agent that you use determines the classes that are available for you to create tests with.

As a best practice, we recommend using xBrowser rather than the Web application because xBrowser uses the Open Agent and dynamic object recognition.

We recommend recording test cases using dynamic object recognition. Then, replay the tests at your convenience.

JavaScript Support for the Open Agent

With the Open Agent, you can use `ExecuteJavaScript` to test anything that uses JavaScript. You can:

- Call any function already contained in a document.
- Inject new functions into a document and call them.
- Trigger Document Object Model (DOM) events, such as calling `onmouseover` directly for an element.
- Modify the DOM tree.

JavaScript Support for the Classic Agent

If you use the Classic Agent, you can test JavaScript using the following methods:

- `ExecLine`
- `ExecMethod`
- `ExecScript`

Oracle Forms Support

This functionality is supported only if you are using the Open Agent.

Silk Test Classic provides built-in support for testing applications that are based on Oracle Forms.



Note: For some controls, Silk Test Classic provides only low-level recording support.

For information on the supported versions and browsers for Oracle Forms, refer to the [Release Notes](#).

Prerequisites for Testing Oracle Forms

To test an application that is built with Oracle Forms, the following prerequisites need to be fulfilled:

- The next-generation Java Plug-In needs to be enabled. This setting is enabled by default. You can change the setting in the **Java Control Panel**. For additional information on the next-generation Java Plug-In, refer to the Java documentation.
- To prevent Java security dialogs from displaying during a test run, the Applet needs to be signed.
- Micro Focus recommends enabling the `Names` property. When this property is enabled, the Oracle Forms runtime exposes the internal `name`, which is the name that the developer of the control has specified for the control, as the `Name` property of the control. Otherwise, the `Name` property will hold a calculated value, which usually consists of the class name of the control plus an index. This enables Silk Test Classic to generate stable locators for controls.


Attributes for Oracle Forms Applications

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Supported attributes for Oracle Forms include:

- `priorlabel`: Helps to identify text input fields by the text of its adjacent label field. Every input field of a form usually has a label that explains the purpose of the input. For controls that do not have a caption, the attribute `priorlabel` is automatically used in the locator. For the `priorlabel` value of a control, for example a text input field, the caption of the closest label at the left side or above the control is used.

- name
- accessibleName

 **Note:** Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Classes in Object-Oriented Programming Languages

Classes are the core of object-oriented programming languages, such as Java. Applets or applications developed in Java are built around objects, which are reusable components of code that include methods and properties. Methods are tasks that can be performed on objects. Properties are characteristics of an object that you can access directly.

Each object is an instance of a class of objects. GUI objects in Java, for example, may belong to such classes as `Menu`, `Dialog`, and `Checkbox`. Each class defines the methods and properties for objects that are part of that class. For example, the `JavaAwtCheckBox` class defines the methods and properties for all Java Abstract Windowing Toolkit check boxes. The methods and properties defined for `JavaAwtCheckboxes` work only on these check boxes, not on other Java objects.

Configuring Silk Test Classic to Test Java

This section describes how to configure Silk Test Classic to test Java applications.

Prerequisites for Testing Java Applications

To test...	Install...
standalone Java applications	JDK/JRE
Java applets	JDK, supported browser, and plug-in (if necessary)
Java applets using the Java Applet Viewer	JDK and plug-in

 **Note:**

- When you are using the Classic Agent, Java support is configured automatically when you use **Enable Extensions** in the **Basic Workflow** bar.
- When you are using the Open Agent, Java support is configured automatically when you use **Configure Applications** in the **Basic Workflow** bar.
- You can use the **Basic Workflow** bar to configure your application or applet or manually configure Java Support. If you choose to manually configure Java support, you may need to change the `CLASSPATH` environment variable. For JVM/JRE 1.2 or later, you must also copy the applicable Silk Test Classic .jar file to the `lib\ext` folder of your JVM/JRE.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Enabling Java Support

There are several ways to enable Java support for testing standalone Java applications. Pick the scenario that fits your runtime environment and testing needs.

Scenario	How to enable Java support
You need to test your application on the 32-bit Windows host machine using a JVM that is invoked from a <code>java.exe</code> , <code>jre.exe</code> , <code>jrew.exe</code> , or <code>vcafe.exe</code> executable, including: <ul style="list-style-type: none"> • JDK/JRE 	Enable the default Java application.

Scenario	How to enable Java support
<ul style="list-style-type: none"> Symantec Visual Café (only if you are using the Classic Agent) <p>You need to test your application on a remote 32-bit Windows machine using a JVM that is invoked from <code>ajava.exe</code>, <code>jre.exe</code>, <code>jrew.exe</code>, or <code>vcafe.exe</code> executable.</p> <p>You need to test your application on the 32-bit Windows host machine using a JVM that is not invoked from a <code>java.exe</code>, <code>jre.exe</code>, <code>jrew.exe</code>, or <code>vcafe.exe</code> executable.</p> <p>You need to test your application on a remote 32-bit Windows machine using a JVM that is not invoked from a <code>java.exe</code>, <code>jre.exe</code>, <code>jrew.exe</code>, or <code>vcafe.exe</code> executable.</p>	<p>Install Silk Test Classic on your remote machine and enable the default Java application on your host machine.</p> <p>Enable a new Java application.</p> <p>Install Silk Test Classic on your target machine and enable a new Java application.</p>

Configuring Silk Test Classic Java Support for the Sun JDK

When you are using the Classic Agent, Java support is configured automatically when you use **Enable Extensions** in the **Basic Workflow** bar. When you are using the Open Agent, Java support is configured automatically when you use **Configure Applications** in the **Basic Workflow** bar. We recommend that you use the basic workflow bar to configure your application or applet, but it is also possible to manually configure Java support.

If you incorrectly alter files that are part of the JVM extension, such as the `accessibility.properties` file, in the `Java\lib` folder, or any of the files in the `jre\lib\ext` directory, such as `SilkTest_Java3.jar`, unpredictable behavior may occur. There are two methods for configuring Silk Test Classic Java Support:

- Manually configuring Silk Test Classic Java support.
- Configuring Standalone Java Applications and Java Applets.

Manually Configuring Silk Test Classic Java Support

If you want to enable Java support manually, or if the **Basic Workflow** does not support your configuration, perform the following tasks:

- | | |
|---|---|
| If you are using the Classic Agent | Click Options > Extensions to open the Extensions dialog box and enable Java applet or application support by checking or un-checking the Java check box for your application. The Java check box can be checked or un-checked for a specific application or applet. If you check or un-check this check box for one extension, it is checked or un-checked for all. |
| If you are using the Open Agent | Click Options > Application Configurations to open the Edit Application Configuration and add a standard test configuration for your Java application. |

Configuring Standalone Java Applications and Java Applets

In order for Silk Test Classic to recognize Java controls, you may need to change the `CLASSPATH` environment variable. For JVM/JRE 1.3 or later, you must also copy the applicable `SilkTest.jar` file to the `lib\ext` folder of your JVM/JRE. The `SilkTest.jar` file is located in the `<SilkTest Install Directory>\JavaEx` directory.

- If you are using JVM/JRE 1.3 or later, use `SilkTest_Java3.jar`.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

2. For Java 1.3 or later, you should not set a specific classpath variable – instead, use the default `CLASSPATH=.`. Copy the `SilkTest_Java3.jar` file to the `lib\ext` folder of your JVM/JRE, and remove any previous Silk Test Classic JAR files.
3. In the Silk Test Classic folder, rename the file `access3.prop` to `accessibility.properties` and copy it to the `Java...\lib` folder.
4. Finally, `qapjconn.dll` and `qapjarex.dll` are new DLL files that must be installed in the `Windows\System32` directory.

The Silk Test Classic installer places these files in the `Windows\System32` folder, and also places copies of these files in the `SilkTest\Extend` folder. If the default directory for your library files is in a location other than `Windows\System32`, you must also copy `qapjconn.dll` and `qapjarex.dll` to the alternate location.



Note:

- The Java recorder does not support applets for embedded Internet Explorer browsers(AOL).
- It is not possible, using normal configuration methods, to gain recognition of Java applications that use `.ini` files to set the environment. However, if your application sets the Java library path using the JVM launcher directive `Djava.library.path=< path >`, you can obtain full recognition by copying `qapjarex.dll` and `qapjconn.dll` from the `System32` directory into the location pointed to by the JVM launcher directive.

Java Security Privileges Required by Silk Test Classic

Before reviewing your security privileges, make sure that you have configured Silk Test Classic Java support.

Required security privileges

In order to load the Silk Test Classic Java support files, Silk Test Classic must have the appropriate Java security privileges. At a minimum, Silk Test Classic requires the following abilities:

- Create a new thread.
- Access members of classes loaded by your application.
- Create, read from, and write to file on a local drive.
- Access, connect, listen and send information through sockets.
- Access AWT event queue.
- Access system properties.

For standalone applications, the security policy is set in the `java.security` file which is located in `JRE/lib/security`. By default this file contains the following line:

```
Policy.provider = sun.security.provider.PolicyFile
```

which means that the standard policy file should be used. The standard policy file, `java.policy`, is located in the same folder, `JRE/lib/security`. It contains the following code that gives all necessary permission to any file located in `lib\ext` directory:

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {permission
java.security.AllPermission;};
```

Silk Test Classic has the necessary privileges, if the `SilkTest_Java3.jar` file is in this directory and the JVM runs with the default set of security permissions.

If you have changed the Java security policy

The system administrator can change security policy by starting the JVM with the following option:

```
java -Djava.security.policy=Myown.policy MyApp
```

In this case the custom policy file `Myown.policy` should contain the following lines that grant all permission to classes from the `lib\ext` directory:

```
grant codeBase "file:${java.home}/lib/ext/*" {permission
java.security.AllPermission;};
```

The default `java.policy` may also be changed implicitly, for example, when the application uses an RMI server with the custom `RMISecurityManager` and the client security policy. In cases like these, the client security policy should grant all required permissions to Silk Test Classic by including the code listed above.

In some cases, setting these permissions may not provide Silk Test Classic with the necessary security privileges. The cause of the problem may be that permissions are frame specific. So if Silk Test Classic runs in the context of frames (thread) in which it does not have the necessary permissions, it may fail. In cases like this in which the client does not trust code running in the context of the AWT event thread, you need to set the parameter `ThreadSafe=False` in the `javaex.ini` in the `<Silk Test installation>/extend` directory. This prevents the Silk Test Classic Java code from running in the context of the AWT event thread, preserving permissions granted to Silk Test Classic, but could make the GUI less responsive.

Testing Java Applications and Applets

Silk Test Classic supports testing Java applications that use the Sun JDK. By default, Silk Test Classic uses the Sun JDK with the Classic Agent.

Preparing for Testing Stand-Alone Java Applications and Applets

To prepare for testing stand-alone Java applications using Silk Test Classic:

1. In the **Basic Workflow** bar:

- If you are using the Classic Agent, enable extensions for Java support for application and applet testing.
- If you are using the Open Agent, configure your Java application.

2. If you do not plan to test applets during the session, disable browser support.

3. Identify the custom controls in your Java application.

4. If you are testing Java applications with the Classic Agent, enable the recovery system.

5. Record classes for any custom controls you want to test in a new class include file or in your test frame file.

If Silk Test Classic does not recognize some of your custom objects, see *Recording Classes for Ignored Java Objects*.

6. If you are testing standalone Java applications with the Classic Agent, record window declarations for your Java application, including declarations for any new classes you defined.

Indexed Values in Test Scripts

4Test methods use a 1-based indexing scheme, where the first indexed value is stored in position 1. Native Java methods use a 0-based indexing scheme, where the first indexed value is stored in position 0. This incompatibility can create challenges in coding test scripts that access indexed values using both native methods and 4Test methods.

When to Use 4Test Versus Native Java Controls

Silk Test Classic provides a predefined set of Java classes, including Abstract Windowing Toolkit (AWT) controls, Java Foundation Class (JFC) controls, and Symantec Visual Café controls. To test these controls, you can use their inherited 4Test methods. Inherited methods are the 4Test methods associated with the class from which the control is derived.

For custom Java controls, we provide access to native Java methods, as defined in JDK 1.1.2 or later. You can also access native methods for predefined Java classes.

When both 4Test methods and native methods are available for all controls you want to test, we recommend using 4Test methods in your test scripts. 4Test provides a richer, more efficient set of methods that more closely mirror user interaction with the GUI elements of an application. For the `JavaAwtPushButton`, for example, use the 4Test methods associated with the `PushButton` class.

When you must use native methods for controls that are not supported in 4Test, refer to the Java API documentation to gain a full understanding of how the native method works. For example, 4Test methods and native Java methods use incompatible array indexing schemes so you must use caution when accessing indexed values.



Note: We recommend not to mix 4Test and native methods because of incompatibilities between Java and 4Test.

Predefined Class Definition File for Java

The file `javaex.inc` includes 4Test class definitions for the following controls:

- Abstract Windowing Toolkit (AWT) controls
- Java Foundation Class (JFC) library controls
- Symantec Visual Café Itools controls
- Java-equivalent window controls

We provide these class definitions to help you quickly get started with testing your Java applications. You can record additional classes if you determine that additional controls are necessary to test your application.

The file `javaex.inc` is installed in the `Extend` subdirectory under the directory where you installed Silk Test Classic.

Troubleshooting Java Applications

This section provides solutions for common reasons that might lead to a failure of the test of your standalone Java application or applet. If these do not solve the specific problem that you are having, you can enable your extension manually.

The test of your standalone Java application or applet may fail if the application or applet was not ready to test, the Java plug-in was not enabled properly, if there is a Java recognition issue, or if the Java applet does not contain any Java controls within the **JavaMainWin**.

What Can I Do If the Silk Test Java File Is Not Included in a Plug-In?

If the `SilkTest_Java3.jar` file is not included in the `lib/ext` directory of the plug-in that you are using:

1. Locate the `lib/ext` directory of the plug-in that you are using and check if the `SilkTest_Java3.jar` file is included in this folder.
2. If the `SilkTest_Java3.jar` file is not included in the folder, copy the file from the `javaex` folder of the Silk Test installation directory into the `lib\ext` directory of the plug-in.

What Can I Do If Java Controls In an Applet Are Not Recognized?

Silk Test Classic cannot recognize any Java children within an applet if your applet contains only custom classes, which are Java classes that are not recognized by default, for example a frame containing only an image. For information about mapping custom classes to standard classes, see *Mapping Custom Classes to Standard Classes*. Additionally, you have to set the Java security privileges that are required by Silk Test Classic.

Supported Java Classes

We provide 4Test definitions in our class definition file for the following Java classes:

- Abstract Windowing Toolkit (AWT) classes
- Java Foundation Class (JFC) library classes
- Symantec Visual Café Itools classes (only for the Classic Agent)
- Java-equivalent window classes

Each of these predefined classes inherits 4Test properties and methods, which are referenced in the class descriptions in this Help. Not all inherited methods have been implemented for Java controls.

You can also access the native methods of the supported classes by removing the 4Test definition and re-recording the class.

The only assumption that the Java extension makes about the implementation of the Java classes in an AUT is that the classes do not violate the standard Swing or AWT models. The Java extension should be able to recognize and manipulate a Java class in an application, as long as the class extends one of the components that the Java extension supports, and any customization does not violate the API of that component. For example, changing a method from public to private violates the API of the component.

Predefined Java-Equivalent Window Classes

The following 4Test classes are provided for testing Java-equivalent window controls:

Classic Agent	Open Agent
JavaApplet	AppletContainer
JavaDialogBox	AWTDialog JDialog
JavaMainWin	AWTFrame JFrame

Predefined AWT Classes

The following 4Test classes are provided for testing Abstract Windowing Toolkit (AWT) controls:

Classic Agent	Open Agent
JavaAwtCheckBox	AWTCheckBox
JavaAwtListBox	AWTList
JavaAwtPopupList	AWTChoice
JavaAwtPopupMenu	No corresponding class.
JavaAwtPushButton	AWTPushButton
JavaAwtRadioButton	AWTRadioButton
JavaAwtRadioList	No corresponding class.
JavaAwtScrollBar	AWTScrollBar
JavaAwtStaticText	AWTLabel
JavaAwtTextField	AWTTextField AWTTextArea

Predefined JFC Classes

The following 4Test classes are provided for testing Java Foundation Class (JFC) controls:

Classic Agent	Open Agent
JavaJFCCheckBox	JCheckBox
JavaJFCCheckBoxMenuItem	JCheckBoxMenuItem
JavaJFCChildWin	No corresponding class.
JavaJFCComboBox	JComboBox
JavaJFCImage	No corresponding class.
JavaJFCListBox	JList
JavaJFCMenu	JMenu
JavaJFCMenuItem	JMenuItem
JavaJFCPageList	JTabbedPane
JavaJFCPopUpList	JList
JavaJFCPopupMenu	JPopupMenu
JavaJFCProgressBar	JProgressBar
JavaJFCPushButton	JButton
JavaJFCRadioButton	JRadioButton
JavaJFCRadioButtonMenuItem	JRadioButtonMenuItem
JavaJFCRadioList	No corresponding class.
JavaJFCScale	JSlider
JavaJFCScrollBar	JScrollBar JHorizontalScrollBar JVerticalScrollBar
JavaJFCSeparator	JComponent
JavaJFCStaticText	JLabel
JavaJFCTable	JTable
JavaJFCTextField	JTextField JTextArea
JavaJFCToggleButton	JToggleButton
JavaJFCToolBar	JToolBar
JavaJFCTreeView	JTree
JavaJFCUpDown	JSpinner

Invoking Java Applications and Applets

This section describes how you can invoke Java applications and applets.

Invoking Java Applets

To invoke the Java applet from within a supported browser, perform the following tasks:

- If you are using the Classic Agent, configure Silk Test Classic for Java support and enable the Java extension.
- If you are using the Open Agent, configure the application.

Invoking JRE Applications

Once you set CLASSPATH for testing standalone Java applications, you are ready to invoke your application using the Java Runtime Environment (JRE).



Note: The JRE ignores the CLASSPATH environment variable. As a result, you must invoke JRE applications with command line arguments to pick up the value of CLASSPATH.

The following table describes the commands you can use:

Command	Description
-cp	Searches first through directories and files specified, then through standard JRE directories.
-classpath	Ignores the value of your CLASSPATH environment variable. You must specify a complete search path on the command line. Does not search the standard JRE directories.

To invoke JRE applications using -cp

Enter the following command:

```
jre -cp %CLASSPATH%;<other directories, if any> <name of application>
```

Example

Assuming your CLASSPATH is set to the complete search path including the Java support path, you would launch the application MyJREapp by entering:

```
jre -cp %CLASSPATH% MyJREapp
```

Invoking JRE Applications Using -classpath

To invoke JRE applications using -classpath, enter the following command:

```
jre -classpath <Java support path>;<other directories, if any>
```

Example

Assuming you installed Silk Test Classic in the default directory c:\Program Files\Silk\SilkTest, you are using JRE 1.1.5 as your Java Virtual Machine (JVM), and your CLASSPATH contains only the Java support path, you would launch the application MyJREapp by entering:

```
Java -classpath c:\Progra~1\Silk\SilkTest\JavaEx  
\SilkTest_Java3.jar MyJREapp
```

invokeMethods Example: Draw a Line in a Text Field

To draw a line in a multiline text field, you need to access a graphics object inside the text field by calling the following methods in Java:

```
main()  
{  
    TextField multiLine = ...; // get reference to multiline text field  
    Graphics graphObj = multiLine.getGraphics();  
    graphObj.drawLine(10, 10, 20, 20);  
}
```

However, you cannot call the above sequence of methods from 4Test because Graphics is not 4Test-compatible. Instead, you can insert the invokeMethods prototype in the TextField class declaration, then

add `invokeMethods` by hand to your test script to draw a line in the Graphics object nested inside the multiline text field, as shown in this `4Test` function:

```
DrawLineInTextField()
MyDialog.Invoke() // Invoke Java dialog that contains the text field
MyDialog.TheTextField.invokeMethods ({"getGraphics", "drawLine"}, {}, {10,
10, 20, 20})
```

In this code, the following methods are called in Java:

- `getGraphics` is invoked on the multiline text field `TheTextField` with an empty argument list, returning a Graphics object.
- `drawLine` is invoked on the Graphics object, to draw a line starting from (x,y) coordinates (10,10) and continuing to (x,y) coordinates (20,20).

Accessing Java Objects and Methods

This section describes how you can access Java objects and methods.

Accessing Nested Java Objects

Sometimes you cannot retrieve `4Test`-compatible information about a Java control with a single call to a `4Test` method; instead, you need to call several nested methods, each returning an intermediate object to be passed to the next method. If any of these methods returns intermediate results that are not `4Test`-compatible, you will not be able to perform these nested calls from `4Test`.

You can use the following methods to access nested Java objects:

Method	Agent	What it does
<code>InvokeJava</code>	Classic Agent	This method allows you to invoke a Java class from <code>4Test</code> for manipulating a nested Java object.
<code>invokeMethods</code>	Classic Agent Open Agent	Allows you to perform nested calls inside Java, even if intermediate results are not <code>4Test</code> -compatible. You can call <code>invokeMethods</code> for any Java object as long as you add the <code>invokeMethods</code> prototype inside the object's class declaration.

Calling Nested Methods

Sometimes you cannot retrieve `4Test`-compatible information about a Java control with a single call to a `4Test` method; instead, you need to call several nested methods, each returning an intermediate object to be passed to the next method. If any of these methods returns intermediate results that are not `4Test`-compatible, you will not be able to perform these nested calls from `4Test`.

You can use the following methods to call nested methods:

Method	Agent	What it does
<code>InvokeJava</code>	Classic Agent	This method allows you to invoke a Java class from <code>4Test</code> for manipulating a nested Java object.
<code>invokeMethods</code>	Classic Agent Open Agent	Allows you to perform nested calls inside Java, even if intermediate results are not <code>4Test</code> -compatible. You can call <code>invokeMethods</code> for any Java object as long as you add the <code>invokeMethods</code> prototype inside the object's class declaration.

Example: How to add an `invokeMethods` prototype to your script

This example shows how to add an `invokeMethods` prototype inside the declaration for a `JavaAwtListBox` in `javaex.inc`.

```
winclass JavaAwtListBox : ListBox
tag "[JavaAwtListBox]"
```

```
setting MultiTags = {TAG_CAPTION}

obj AnyType invokeMethods(list of Strings strA, List of List
of Anytype anyaa)
```

Testing Java Scroll Panes

A scroll pane in Java is a container that holds a single child component. If the scroll pane is smaller than the child component, you can scroll vertically and horizontally to see all parts of that component.

The state of the scroll bars in a scroll pane is managed by internal objects that implement the `Adjustable` interface. To manipulate the scroll bars, you must first get an `Adjustable` object, and then use `Adjustable` and scroll bar methods to move them.

To test scroll bars in a scroll pane, use `invokeMethods`, a method that allows you to perform nested calls inside Java to access `Adjustable` objects.

Frequently Asked Questions About Testing Java Applications

This section provides answers to frequently asked questions about classpath and testing Java applications and applets.

Can I Invoke Java Code from 4Test Scripts?

- If you are using the Classic Agent, you can invoke Java code from 4Test scripts using the method `InvokeJava`.
- You can invoke Java code from 4Test scripts using the method `invokeMethods`, for both the Classic Agent and the Open Agent.

Can I Test JavaScript Objects?

With the Classic Agent, you can use `InvokeJava` to access methods for testing JavaScript objects, if these objects reside on a Web page that contains an applet.

With the Open Agent, you can use `ExecuteJavaScript` to test anything that uses JavaScript.

Can I test JDeveloper and JBuilder with Silk Test Classic?

When using the Open Agent, Silk Test Classic supports test JDeveloper and JBuilder out of the box, just like any other supported Java AWT/Swing application.

Can I Use the Java Plug-In to Test Applets Outside My Browsers Native JVM?

For testing purposes, you can use the Java plug-in to run applets outside the native Java virtual machine of your browser.

How Can I Record AWT Menus?

You cannot use the `JavaAwtPopupMenu` class to record AWT menus. It is available for playback only. You must manually script any interaction with AWT menus.

How Do I Decide Whether to Use 4Test Methods or Native Methods?

For information on when to use 4Test methods or native methods, see *When to Use 4Test Versus Native Java Controls*.

Why Do I See so Many Java CustomWin Objects?

Objects that do not belong to any of our predefined Java classes are custom controls, which are identified as `CustomWin` objects by Silk Test Classic. Most Java applications and applets use many custom controls to fine tune functionality and the user interface.

To manipulate a custom Java object for testing, you do not need to write your own extensions. Instead, you can use the object's own native methods and properties. Our Java support lets you access native methods and properties, by recording classes for custom controls.

Why Do I Need to Disable the Classpath if I have Java Installed but Am not Testing It?

If you are not testing Java but do have Java installed, we recommend that you disable the classpath before using Silk Test Classic. If you do not disable the classpath, Silk Test Classic checks for a Java classpath every time you run a test plan. To disable the classpath during the Silk Test Classic installation, select **None** on the **Java** dialog box. To verify that you have disabled the classpath, verify that the path to the Java extension is disabled in the *Java* variable, which is stored in the system variables.

For example, to verify that the path to the Java extension is disabled on Microsoft Windows 7, perform the following steps:

1. Click **Start > Control Panel**.
2. In the **Control Panel**, click **System and Security**.
3. In the **System and Security** pane, click **System**.
4. In the **System** pane, click **Advanced System Settings**.
5. In the **System Properties** dialog box, click **Environment Variables**.
6. In the **System variables** area of the **Environment Variables** dialog box, select the *Java* variable.
7. Disable the path to the Java extension, by placing an underscore at the beginning of the path.

Testing Java SWT and Eclipse Applications with the Open Agent

Silk Test Classic provides built-in support using the Basic Workflow for testing applications that use widgets from the Standard Widget Toolkit (SWT) controls. When you configure a Java SWT/RCP application, Silk Test Classic automatically provides support for testing standard Java SWT/RCP controls.

Silk Test Classic supports:

- Testing Java SWT controls embedded in Java AWT/Swing applications as well as Java AWT/Swing controls embedded in Java SWT applications.
- Standalone SWT applications that use the EXT or CLASSPATH configuration.
- Testing Java SWT applications that use the IBM JDK or the Sun JDK.
- Any Eclipse-based application that uses SWT widgets for rendering. Silk Test Classic supports both Eclipse IDE-based applications and RCP-based applications.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Java SWT applications support dynamic object recognition. When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Existing test cases that use dynamic object recognition without locator keywords in an INC file will continue to be supported. You can replay these tests, but you cannot record new tests with dynamic object recognition without locator keywords in an INC file.

Using custom class attributes becomes even more powerful when it is used in combination with dynamic object recognition.

For a complete list of the widgets available for SWT testing, see *Supported SWT Widgets for the Open Agent*.

For a complete list of the record and replay controls available for Java SWT testing, view the `SWT.inc` and `JavaSWT.inc` file. To access the `JavaSWT.inc` file that is used with the Open Agent, navigate to the `<SilkTest directory>\extend\JavaSWT` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\extend\JavaSWT\JavaSWT.inc`. To access the `SWT.inc` file, navigate to the `<SilkTest directory>\extend` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\extend\SWT.inc`.

Suppressing Controls (Open Agent)

This functionality is supported only if you are using the Open Agent.

To simplify the object hierarchy and to shorten the length of the lines of code in your test scripts and functions, you can suppress the controls for certain unnecessary classes in the following technologies:

- Win32.
- Java AWT/Swing.
- Java SWT/Eclipse.
- Windows Presentation Foundation (WPF).

For example, you might want to ignore container classes to streamline your test cases.

To suppress specific controls:

1. Click **Options > Recorder**. The **Recording Options** dialog box opens.
2. Click the **Transparent Classes** tab.
3. Type the name of the class that you want to ignore during recording and playback into the text box.

If the text box already contains classes, add the new classes to the end of the list. Separate the classes with a comma. For example, to ignore both the `AOL_Toolbar` and the `_AOL_Toolbar` class, type `AOL Toolbar, _AOL_Toolbar` into the text box.

The `OPT_TRANSPARENT_CLASSES` option is set to true for these classes.

4. Click **OK**. The `OPT_TRANSPARENT_CLASSES` option is set to true for these classes, which means the classes are added to the list of the classes that are ignored during recording and playback.

Custom Attributes

This functionality is supported only if you are using the Open Agent.

Add custom attributes to a test application to make a test more stable. You can use custom attributes with the following technologies:

- Java SWT
- Swing
- WPF
- xBrowser
- Windows Forms
- SAP

For example, in Java SWT, the developer implementing the GUI can define an attribute (for example, `silkTestAutomationId`) for a widget that uniquely identifies the widget in the application. A tester using

Silk Test Classic can then add that attribute to the list of custom attributes (in this case, `silkTestAutomationId`), and can identify controls by that unique ID. Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index will change whenever another widget is added before the one you have defined already.

If more than one object is assigned the same custom attribute value, all the objects with that value will return when you call the custom attribute. For example, if you assign the unique ID, 'loginName' to two different fields, both fields will return when you call the `loginName` attribute.

First, enable custom attributes for your application and then create the test.

Recording tests that use dynamic object recognition

Using custom class attributes becomes even more powerful when it is used in combination with dynamic object recognition. For example, if you create a button in the application that you want to test using the following code:

```
Button myButton = Button(parent, SWT.NONE);
myButton.setData("SilkTestAutomationId", "myButtonId");
```

To add the attribute to your XPath query string in your test case, you can use the following query:

```
Window button = Desktop.Find("//
PushButton[@SilkTestAutomationId='myButton']")
```

Locator Attributes for Java SWT Controls

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Silk Test Classic supports the following locator attributes for Java SWT controls:

- caption
- all custom object definition attributes



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Dynamically Invoking Java Methods

This functionality is supported only if you are using the Open Agent.

You can call methods, retrieve properties, and set properties on controls that Silk Test Classic does not expose by using the dynamic invoke feature. This feature is useful for working with custom controls and for working with controls that Silk Test Classic supports without customization.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList()` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty()` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList()` method.



Note: Typically, most properties are read-only and cannot be set.

Supported Methods and Properties

The following methods and properties can be called:

- Methods and properties that Silk Test Classic supports for the control.
- All public methods of the SWT, AWT, or Swing widget.
- If the control is a custom control that is derived from a standard control, all methods and properties from the standard control can be called.

Supported Parameter Types

The following parameter types are supported:

All built-in Silk Test Classic types. Silk Test Classic types includes primitive types (such as boolean, int, string), lists, and other types (such as Point and Rect).

Enum types. Enum parameters must be passed as string. The string must match the name of an enum value. For example, if the method expects a parameter of the enum type, `java.sql.ClientInfoStatus` you can use the string values of `REASON_UNKNOWN`, `REASON_UNKNOWN_PROPERTY`, `REASON_VALUE_INVALID`, or `REASON_VALUE_TRUNCATED`.

Example

A custom calculator control has a `Reset` method and an `Add` method, which performs an addition of two numbers. You can use the following code to call the methods directly from your tests:

```
customControl.Invoke("Reset")  
REAL sum = customControl.DynamicInvoke("Add", {1,2})
```

Java SWT Classes for the Open Agent

Troubleshooting Java SWT and Eclipse Applications

Some SWTTree methods do not replay with low-level playback

When using low-level playback, some SWTTree methods, for example `Expand` and `Collapse`, do not replay.

To solve this problem, set the replay mode to **Default**. For additional information, see *Setting Replay Options for the Open Agent*.

Selecting a non-visible node in an SWTTree

When using low-level playback, Silk Test Classic cannot interact with non-visible nodes in an SWTTree.


To solve this problem, set the replay mode to **Default**. For additional information, see *Setting Replay Options for the Open Agent*.


Testing Mobile Applications

Silk Test Classic enables you to automatically test your native mobile applications (apps) and mobile web applications. Automatically testing your mobile applications with Silk Test Classic provides the following benefits:

- It can significantly reduce the testing time of your mobile applications.

- You can create your tests once and then test your mobile applications on a large number of different devices and platforms.
- You can ensure the reliability and performance that is required for enterprise mobile applications.
- It can increase the efficiency of QA team members and mobile application developers.
- Manual testing might not be efficient enough for an agile-focused development environment, given the large number of mobile devices and platforms on which a mobile application needs to function.

 **Note:** To test native mobile applications or hybrid applications with Silk Test Classic, you require a native mobile license. For additional information, see [Licensing Information](#).

 **Note:** Silk Test Classic provides support for testing mobile apps on both Android and iOS devices.

For information on the supported operating system versions and the supported browsers for testing mobile applications, refer to the [Release Notes](#).

Android

Silk Test Classic enables you to test a mobile application on an Android device or an Android emulator.

Prerequisites for Testing Mobile Applications on Android

Before you can test a mobile application (app) on an Android device or on an Android emulator, ensure that the following prerequisites are met:

- Ensure that Java is installed on the machine on which Silk Test Classic is running, and that the path to Java is added to the *Path* environment variable. Silk Test Classic requires either a JRE or a JDK for mobile testing. If Java is not installed, add `C:\Program Files (x86)\Silk\SilkTest\ng\jre\bin` to the *Path*.
- If you have created your own hybrid app by adding a web view to a native mobile app, add the following code to the app to make your app testable with Silk Test Classic:


```
WebView.setWebContentsDebuggingEnabled(true);
webView.getSettings().setJavaScriptEnabled(true);
```
- Silk Test Classic does not support showing a live view of the device screen for Android 4.4. Micro Focus recommends using Android 5 or later.

Testing Mobile Applications on Android

To test a mobile application on a physical Android device or on an Android emulator, perform the following tasks:

1. Ensure that you have met the prerequisites for testing mobile applications on Android.
For additional information, see [Prerequisites for Testing Mobile Applications on Android](#).
2. If you want to test the mobile application on an Android emulator, configure the emulator settings for Silk Test Classic.
For additional information, see [Configuring the Android Emulator for Silk Test](#).
3. Start the Android emulator or connect the device to the machine on which Silk Test Classic is installed.
4. If you want to test the mobile application on a physical Android device that you are using for the first time on this machine, install the appropriate Android USB Driver on the machine.
For additional information, see [Installing a USB Driver](#).
5. If you want to test the mobile application on a physical Android device, enable USB-debugging on the Android device.
For additional information, see [Enabling USB-Debugging](#).
6. Create a Silk Test Classic project for your mobile application.
7. Create a test for your mobile application.

8. Record the actions that you want to execute in the test. When you start the **Recording** window, the **Select Application** dialog box opens.
9. To test a mobile web application:
 - a) Select the **Web** tab.
 - b) Select the mobile browser that you want to use.
 - c) Specify the web page to open in the **Enter URL to navigate** text box.

10. To test a native mobile application or a Hybrid application:



Note: To test native mobile applications or hybrid applications with Silk Test Classic, you require a native mobile license. For additional information, see *Licensing Information*.

- a) Select the **Mobile** tab.
- b) Select the mobile device, on which you want to test the app, from the list.
- c) Click **Browse** to select the app file or enter the full path to the app file into the **Mobile app file** text field.

Silk Test Classic supports HTTP and UNC formats for the path.

Silk Test Classic installs the app on the mobile device or emulator.

11. Click **OK**.

An Android device or emulator must not be screen-locked during testing. To keep the device awake while it is connected to a machine, open the settings and tap **Developer Options**. Then check **Stay awake** or **Stay awake while charging**.

12. Use the **Recording** window to record the test against the mobile application.

For additional information, see [Recording Mobile Applications](#).

13. When you have recorded all the actions, stop the recording.

14. Replay the test.

15. Analyze the test results.

Testing Hybrid Applications on Android

Hybrid applications (apps) are apps that are run on the device, like native applications, but are written with web technologies, for example HTML5, CSS, and JavaScript.

Silk Test Classic supports testing hybrid apps that consist of a single web view which is embedded in a native container. A common example of such a hybrid app would be an Apache Cordova application.

Silk Test Classic can detect web views that support Chrome remote debugging. Silk Test Classic can detect web views with either the package *com.android.webview* or the package *com.google.android.webview*, which are the default packages on most Android devices.



Note: Silk Test Classic supports testing hybrid apps on Android 4.4 or later. To test hybrid apps on Android 6.0 or later, Android System WebView version 51 or later is required.

The process for testing a hybrid app on Android is the same as the process for testing a mobile native application. For additional information, see [Testing Mobile Applications on Android](#).

Installing a USB Driver

To connect an Android device for the first time to your local machine to test your mobile applications, you need to install the appropriate USB driver.

The device manufacturer might provide an executable with all the necessary drivers for the device. In this case you can just install the executable on your local machine. If the manufacturer does not provide such an executable, you can install a single USB driver for the device on the machine.

To install the Android USB driver:

1. Download the appropriate driver for your device.

For example, for information on finding and installing a USB driver for a Google Nexus device, see <http://developer.android.com/tools/extras/oem-usb.html>.

2. Connect your Android device to a USB port on your local machine.
3. From your desktop or **Windows Explorer**, right-click **Computer** and select **Manage**.
4. In the left pane, select **Device Manager**.
5. In the right pane, locate and expand **Other device**.
6. Right-click the device name, for example *Nexus 5x*, and select **Update Driver Software**. The **Hardware Update Wizard** opens.
7. Select **Browse my computer for driver software** and click **Next**.
8. Click **Browse** and navigate to the folder to which you have downloaded the USB driver.
9. Select the USB driver.
10. Click **Next** to install the driver.

Enabling USB-Debugging

To communicate with an Android device over the Android Debug Bridge (adb), enable USB debugging on the device.

1. On the Android device, open the settings.
2. Tap **Developer Settings**.

The developer settings are hidden by default. If the developer settings are not included in the settings menu of the device:

- a) Depending on whether the device is a phone or a pad, scroll down and tap **About phone** or **About Pad**.
- b) Scroll down again and tap **Build Number** seven times.

3. In the **Developer settings** window, check **USB-Debugging**.
4. Set the USB mode of the device to **Media device (MTP)**, which is the default setting.
For additional information, refer to the documentation of the device.

Recommended Settings for Android Devices

To optimize testing with Silk Test Classic, configure the following settings on the Android device that you want to test:

- Enable USB-debugging on the Android device. For additional information, see [Enabling USB-Debugging](#)
- An Android device must be connected as a media device to the machine on which the Open Agent is running. The USB mode of the Android device must be set to **Media device (MTP)**.
- An Android device or emulator must not be screen-locked during testing. To keep the device awake while it is connected to a machine, open the settings and tap **Developer Options**. Then check **Stay awake** or **Stay awake while charging**.

Configuring the Android Emulator for Silk Test Classic


When you want to test mobile applications on an Android emulator with Silk Test Classic, you have to configure the emulator for testing:

1. Install the latest version of the Android SDK.
For information on how to install and configure the Android SDK, see [Get the Android SDK](#).
2. Install Android Studio 2.

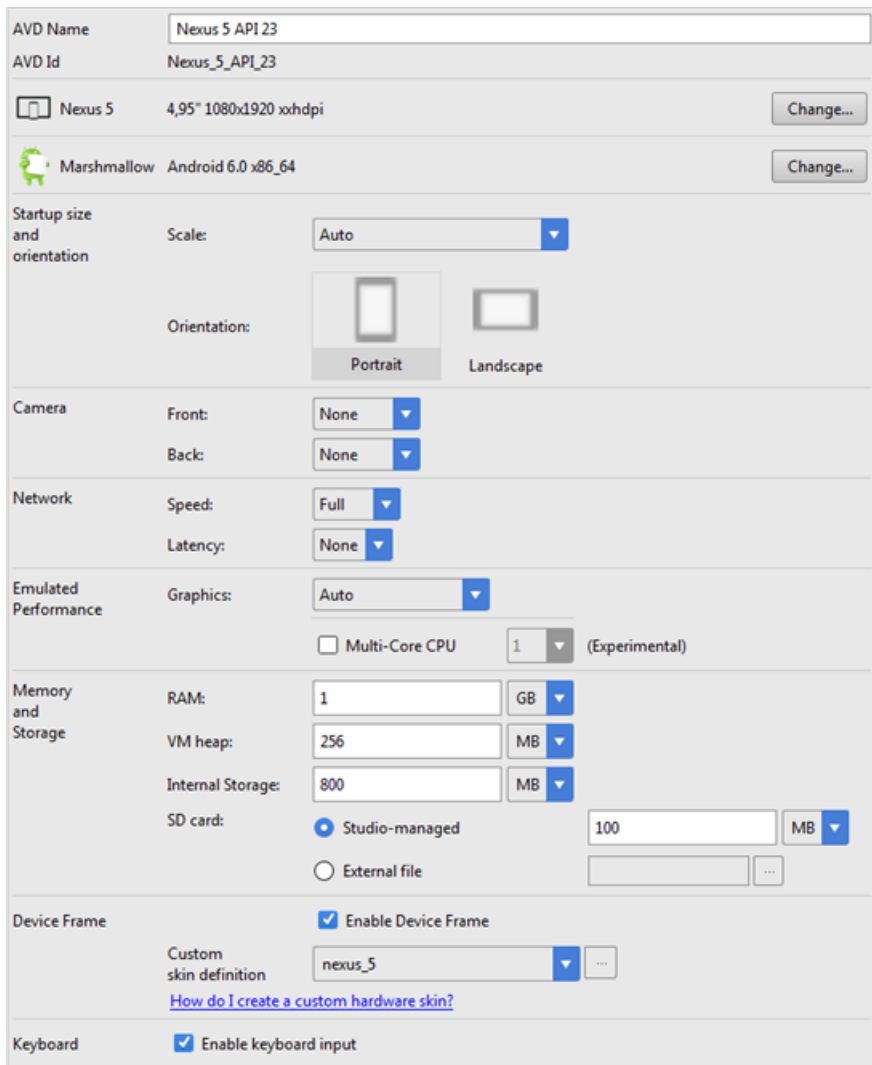


Tip: You can skip installing Android Studio 2 and use the emulator provided with the Android SDK. However, Micro Focus recommends installing Android Studio 2 for improved emulator performance. The remaining steps in this topic require Android Studio 2 to be installed.

3. From Android Studio 2, start the **AVD Manager**.
4. Click **Create Virtual Device**.
5. Select a virtual device.
6. Click **Next**.
7. Download and select a system image of Android that includes Google APIs.
8. Click **Next**.
9. Configure the virtual device according to your requirements.
10. Click **Show Advanced Settings**.
11. Adjust the RAM size and the heap space used by the emulator to an amount that is manageable by your machine.

 **Tip:** Micro Focus recommends using at least 1 GB RAM and 256 MB heap space.

12. Select **Auto** from the list in the **Emulated Performance** area.



The screenshot shows the 'Show Advanced Settings' dialog for an AVD named 'Nexus 5 API 23'. The 'Emulated Performance' section is expanded, showing 'Graphics' set to 'Auto' and 'Multi-Core CPU' set to '1 (Experimental)'. The 'Memory and Storage' section shows 'RAM' set to '1 GB', 'VM heap' set to '256 MB', and 'Internal Storage' set to '800 MB'. The 'Device Frame' section shows 'Enable Device Frame' checked and 'Custom skin definition' set to 'nexus_5'. The 'Keyboard' section shows 'Enable keyboard input' checked.

13. Click **Finish**.

iOS


Silk Test Classic enables you to test a mobile application on an iOS device or an iOS Simulator.

Prerequisites for Testing Mobile Applications on iOS

Before you can test a mobile application (app) on an iOS device or on an iOS Simulator, ensure that the following prerequisites are met:

- Ensure that Java is installed on the machine on which Silk Test Classic is running, and that the path to Java is added to the *Path* environment variable. Silk Test Classic requires either a JRE or a JDK for mobile testing. If Java is not installed, add `C:\Program Files (x86)\Silk\SilkTest\ng\jre\bin` to the *Path* and restart the machine.
- The iOS device is connected to a Mac or the desired iOS Simulator image is installed on a Mac.
- The information service is installed on the Mac. For additional information, see [Installing the Silk Test Infoservice on a Mac](#).
- A supported version of Xcode is installed on the Mac. For information about the supported versions of Xcode, refer to the [Release Notes](#).
- Silk Test Classic is installed on a Windows machine.
- The Mac is located in the same network as the Windows machine and is added as a remote location to the Windows machine.
- To test a native mobile app on an iOS device, ensure that the `.ipa` file of your app has been signed with a developer account. For additional information, see [Preparing an iOS App for Testing](#).
- To test a native mobile app on an iOS Simulator, ensure that the app has been zipped. For additional information, see [Testing Native Mobile Applications on an iOS Simulator](#).
- To test a native mobile app on both an iOS device and an iOS Simulator, ensure that both the signed `.ipa` file and the zipped `.app` directory are located in the same folder.
- If you want to test a native mobile app, ensure that the ID of the iOS device is associated with the developer profile which was used to sign the app.
- If you want to test a mobile web application, use Xcode to build the `SafariLauncher.app`. The files that are required to build the app are located under `/Applications/Silk/Mobile/osx/Appium/node_modules/appium/node_modules/appium-ios-driver/node_modules/safari-launcher`. The output must be generated to the folder `/Applications/Silk/Mobile/osx/Appium/node_modules/appium/node_modules/appium-ios-driver/build/SafariLauncher`.
- The iOS device must not fall into sleep mode during testing. To turn the screen lock and password off, select **Settings > General > Passcode Lock**.
- The Mac should not switch off the screen during testing, otherwise the **Playback Status** dialog box will not display anything.
- If you want to test a mobile application on an iOS Simulator, the Mac, on which the iOS Simulator is installed, must not have the display sleep activated during testing.
- To test a native mobile app on a physical iOS device, enable the UI automation on the device. For additional information, see [Preparing an iOS Device for Testing](#).
- To test a mobile web application with Apple Safari on a physical iOS device, activate the **Web Inspector**. For additional information, see [Preparing an iOS Device for Testing](#).
- Micro Focus recommends using iOS devices which have a Lightning connector. Silk Test Classic does not support showing a live view of the device screen for iOS devices that are not connected to a Mac through a Lightning cable.

Testing Native Mobile Applications on a Physical iOS Device

 **Note:** To test native mobile applications or hybrid applications with Silk Test Classic, you require a native mobile license. For additional information, see [Licensing Information](#).

To test a native mobile application (app) or a hybrid application on a physical iOS device, perform the following tasks:

1. Prepare the iOS device for testing.
For additional information, see [Preparing an iOS Device for Testing](#).

2. Prepare the app for testing.

For additional information, see [Preparing an iOS App for Testing](#).

3. Add the Mac, to which the iOS device is connected, as a remote location to the Windows machine on which Silk Test is installed.

For additional information, see [Editing Remote Locations](#).



Note: At any given point in time, each user on a Mac can only test on one iOS device that is connected to the Mac or on one iOS Simulator that is running on the Mac. To test on multiple iOS devices or iOS Simulators on the same Mac, a user session on the Mac is required for each device and Simulator. For additional information, see [Testing Multiple iOS Simulators on the Same Mac](#).

4. Create a Silk Test Classic project for your mobile application.
5. Create a test for your mobile application.
6. Record the actions that you want to execute in the test. When you start the **Recording** window, the **Select Application** dialog box opens.
7. Select the **Mobile** tab.
8. Select the mobile device, on which you want to test the app, from the list.
9. Click **Browse** to select the app file or enter the full path to the app file into the **Mobile app file** text field. Silk Test Classic supports HTTP and UNC formats for the path. Silk Test Classic installs the app on the mobile device.
10. Click **OK**.

An iOS device or Simulator must not fall into sleep mode during testing. To turn the screen lock and password off, select **Settings > General > Passcode Lock**.

11. When you have recorded all actions, stop recording.
12. Replay the test.
13. Analyze the test results.



Note: To test a native mobile app on both an iOS device and an iOS Simulator, ensure that both the signed `.ipa` file and the zipped `.app` directory are located in the same folder.

Testing Native Mobile Applications on an iOS Simulator



Note: To test native mobile applications or hybrid applications with Silk Test Classic, you require a native mobile license. For additional information, see [Licensing Information](#).

To test a native mobile application (app) or a hybrid application on an iOS Simulator, perform the following tasks:

1. In the Xcode project of the app, compile the app for the iOS Simulator.

You can compile the app either from the Xcode UI or from the command line. For example, to compile the app through the command line for an iOS Simulator with iOS 9.2, execute the following command:

```
xcodebuild -sdk iphonesimulator9.2
```

2. Zip up the `.app` directory of the app into a `.zip` file.

By default, the `.app` directory is located in the directory `~/Library/Developer/Xcode/DerivedData`. You can click **File > Project Settings** in Xcode to see into which location the directory is stored.

3. Add the Mac, on which the iOS Simulator is installed, as a remote location to the Windows machine on which Silk Test Classic is installed.

For additional information, see [Editing Remote Locations](#).



Note: You can only test on one iOS Simulator that is installed on a Mac. Multiple Silk Test Classic users cannot simultaneously test on multiple iOS Simulators that are installed on the same Mac.

4. Create a Silk Test Classic project for your mobile application.

5. Create a test for your mobile application.
6. Record the actions that you want to execute in the test. When you start the **Recording** window, the **Select Application** dialog box opens.
7. Select the **Mobile** tab.
8. Select the iOS Simulator from the list.
9. Click **Browse** to select the zipped app file or enter the full path to the zipped app file into the **Mobile app file** text field.
Silk Test Classic supports HTTP and UNC formats for the path.
Silk Test Classic installs the app on the iOS Simulator.
10. Click **OK**.
An iOS device or Simulator must not fall into sleep mode during testing. To turn the screen lock and password off, select **Settings > General > Passcode Lock**.
11. When you have recorded all actions, stop recording.
12. Replay the test.
13. Analyze the test results.



Note: To test a native mobile app on both an iOS device and an iOS Simulator, ensure that both the signed `.ipa` file and the zipped `.app` directory are located in the same folder.

Testing Mobile Web Applications on a Physical iOS Device

To test a mobile web application on a physical iOS device, perform the following tasks:

1. Prepare the iOS device for testing.
For additional information, see [Preparing an iOS Device for Testing](#).
2. Add the Mac, to which the iOS device is connected, as a remote location to the Windows machine on which Silk Test is installed.
For additional information, see [Editing Remote Locations](#).



Note: At any given point in time, each user on a Mac can only test on one iOS device that is connected to the Mac or on one iOS Simulator that is running on the Mac. To test on multiple iOS devices or iOS Simulators on the same Mac, a user session on the Mac is required for each device and Simulator. For additional information, see [Testing Multiple iOS Simulators on the Same Mac](#).

3. Create a Silk Test Classic project for your mobile application.
4. Create a test for your mobile application.
5. Record the actions that you want to execute in the test. When you start the **Recording** window, the **Select Application** dialog box opens.
6. To test a mobile web application:
 - a) Select the **Web** tab.
 - b) Select the mobile browser that you want to use.
 - c) Specify the web page to open in the **Enter URL to navigate** text box.
7. Click **OK**.
An iOS device or Simulator must not fall into sleep mode during testing. To turn the screen lock and password off, select **Settings > General > Passcode Lock**.
8. When you have recorded all actions, stop recording.
9. Replay the test.
10. Analyze the test results.

Testing Mobile Web Applications on an iOS Simulator

To test a mobile web application on an iOS Simulator, perform the following tasks:

1. Add the Mac, to which the iOS device is connected, as a remote location to the Windows machine on which Silk Test is installed.

For additional information, see [Editing Remote Locations](#).



Note: At any given point in time, each user on a Mac can only test on one iOS device that is connected to the Mac or on one iOS Simulator that is running on the Mac. To test on multiple iOS devices or iOS Simulators on the same Mac, a user session on the Mac is required for each device and Simulator. For additional information, see [Testing Multiple iOS Simulators on the Same Mac](#).

2. Create a Silk Test Classic project for your mobile application.
3. Create a test for your mobile application.
4. Record the actions that you want to execute in the test. When you start the **Recording** window, the **Select Application** dialog box opens.
5. To test a mobile web application:
 - a) Select the **Web** tab.
 - b) Select the mobile browser that you want to use.
 - c) Specify the web page to open in the **Enter URL to navigate** text box.
6. Click **OK**.

An iOS device or Simulator must not fall into sleep mode during testing. To turn the screen lock and password off, select **Settings > General > Passcode Lock**.
7. When you have recorded all actions, stop recording.
8. Replay the test.
9. Analyze the test results.

Testing Hybrid Applications on iOS

Hybrid applications (apps) are apps that are run on the device, like native applications, but are written with web technologies, for example HTML5, CSS, and JavaScript.

Silk Test Classic supports testing hybrid apps that consist of a single web view which is embedded in a native container. A common example of such a hybrid app would be an Apache Cordova application.

The process for testing a hybrid app on iOS is the same as the process for testing a mobile native application. For additional information, see [Testing Native Mobile Applications on a Physical iOS Device](#) or [Testing Native Mobile Applications on an iOS Simulator](#).

Before testing a hybrid app on an iOS device, ensure that the **Web Inspector** is activated on the device.

Preparing an iOS Device for Testing



Note: To test native mobile applications or hybrid applications with Silk Test Classic, you require a native mobile license. For additional information, see [Licensing Information](#).

To prepare the iOS device to test mobile applications:

1. Start Xcode on the Mac.
2. Connect the iOS device to the Mac.
3. On the iOS device, click **Settings > Developer**.



Tip: If the Developer menu is not displayed on the iOS device, restart the device and the Mac.

4. Activate **Enable UI Automation**.
5. To test a mobile web application on Apple Safari, click **Settings > Safari > Advanced**.
6. Activate the **Web Inspector**.

Preparing an iOS App for Testing

This functionality is supported only if you are using the Open Agent.

To be able to test a specific iOS app on a specific iOS device with Silk Test Classic, consider the following:

- Test automation is only possible with iOS apps that can be installed manually on specific iOS devices. To be able to sign an iOS app, you require a membership in the *Apple Developer Program*. For additional information, see [Choosing a Membership](#).



Note: You cannot automatically test iOS apps that are created for publication in the App Store, as well as apps that can be installed manually on any iOS device.

- Before you can install and execute an iOS app on a specific iOS device, you have to register the iOS device with your Apple Developer account.
- You have to use Xcode to create an IPA file of the iOS app, which you can then install on the iOS device. To create IPA files for testing on a specific iOS device, members of the Apple Developer Program can use the *Archive* mechanism of Xcode, by using one of the following two options:
 - If you are a member of the *Apple Developer Enterprise Program*, you can use the **Save for Ad Hoc Deployment** option.
 - If you are a member of the Apple Developer Program, but not of the Apple Developer Enterprise Program, you can use the **Save for Development Deployment** option.

For additional information, see [Exporting Your App for Testing \(iOS, tvOS, watchOS\)](#).

To be able to test a specific iOS app on an iOS Simulator with Silk Test Classic, use Xcode to create a ZIP file of the iOS app, which you can then install on the iOS Simulator. For additional information, refer to the Xcode documentation.

Installing the Silk Test Information Service on a Mac



Note: To install the information service on a Mac, you require administrative privileges on the Mac.


This functionality is supported only if you are using the Open Agent.

To create and execute tests against Apple Safari on a Mac, or against mobile applications on an iOS or Android device that is connected to a Mac, install the Silk Test information service (information service) on the Mac, and then use the **Remote Locations** dialog box to connect a Windows machine, on which Silk Test Classic is installed, to the Mac.

To install the information service on a Mac:

1. Ensure that a Java JDK is installed on the Mac.
2. If you want to test mobile applications on an iOS device, ensure that Xcode is installed on the Mac.
3. Access the information service setup file, `SilkTestInformationService<Version>-<Build Number>.pkg`.
 - If you have downloaded the information service setup file while installing Silk Test, open the folder `macOS` in the Silk Test installation directory, for example `C:\Program Files (x86)\Silk\SilkTest`.
 - If you have not downloaded the information service setup file while installing Silk Test, you can download the setup file from [Micro Focus SupportLine](#).
4. Copy the file `SilkTestInformationService<Version>-<Build Number>.pkg` to the Mac.
5. Execute `SilkTestInformationService<Version>-<Build Number>.pkg` to install the information service.
6. Follow the instructions in the installation wizard.
7. When asked for the password, provide the password of the currently signed in Mac user.
8. When Apple Safari opens and a message box asks whether to trust the SafariDriver, click **Trust**.

To complete the installation, the installer logs the current Mac user out. To verify that the information service was installed correctly, log in to the Mac and click on the Silk Test icon in the top-right corner of the screen to see the available devices and browsers.

 **Tip:** If the Silk Test icon does not appear, restart the Mac.

Uninstalling the Silk Test Information Service from a Mac

This functionality is supported only if you are using the Open Agent.

To uninstall the Silk Test information service (information service) from a Mac, for example if you no longer want to execute tests against Apple Safari on the Mac:

1. Create a new shell file, for example `uninstallInfoService.sh`.
2. Type the following code into the new file:

```
#!/bin/sh

if launchctl list | grep com.borland.infoservice ; then
    launchctl unload /Library/LaunchAgents/com.borland.infoservice.plist
    echo "unloading Launch Daemon"
fi

if [ -d "/Applications/Silk" ]
then
    sudo rm -rf /Applications/Silk
fi

if [ -f "/Library/LaunchAgents/com.borland.infoservice.plist" ]
then
    sudo rm /Library/LaunchAgents/com.borland.infoservice.plist
fi

if [ -f "/usr/local/bin/ideviceinstaller" ]
then
    sudo rm /usr/local/bin/ideviceinstaller
fi

exit 0
```

3. In the command line, type `chmod +x uninstallInfoService.sh` to make the shell file executable.
4. Execute the shell file from the command line.

Recommended Settings for iOS Devices

To optimize testing with Silk Test Classic, configure the following settings on the iOS device that you want to test:

- To make the testing reflect the actions an actual user would perform, disable AutoFill and remembering passwords for Apple Safari. Tap **Settings > Safari > Passwords & AutoFill** and turn off the **Names and Passwords** setting.
- The iOS device must not fall into sleep mode during testing. To turn the screen lock and password off, select **Settings > General > Passcode Lock**.

Testing Multiple iOS Devices or Simulators on the Same Mac

By default, Silk Test does not support testing multiple iOS devices or iOS Simulators that are connected to the same Mac. At any given time, any given user can only test on one iOS device or one iOS Simulator that is connected to the Mac. However, you can create multiple user sessions on a Mac to test multiple iOS devices that are connected to the Mac and multiple iOS Simulators that are running on the Mac in parallel.

1. For each iOS device or iOS Simulator, create a user with administrator privileges on the Mac. For example, to test on two iOS devices that are connected to the Mac and two iOS Simulators that are running on the Mac, you require four user sessions on the Mac with administrative privileges.
2. For each user, install the Silk Test information service (information service).

For information on installing the information service, see [Installing the Silk Test Infoservice on a Mac](#).

3. For each user, navigate to the directory `.silk/silktest/conf/` and rename the file `infoservice.properties.sample` to `infoservice.properties`.
4. For each user, edit the `infoservice.properties` file to specify a port that is not in use.
The default port is 22901. Typically, you could increment the default port number by one for each user. For example, if you want to use four user sessions, use the ports 22901, 22902, 22903, and 22904.
5. For each user, logout and login again with the same user to activate the information service with the specified port.



Tip: To use the information service that corresponds to a user session, ensure that the user is logged on to the Mac, for example after a reboot of the Mac.

Testing an Installed App

To test a native mobile app that is already installed on a device, an Emulator, or a Simulator, specify the app in the connection string.

1. Open an existing project that tests a native mobile app.
2. Open the **Edit Application Configuration** dialog box.
3. Replace the existing app in the connection string with one of the following:
 - If you want to test an iOS app, replace the app with the *bundleId*, for example replace `app=MyApp.ipa` with `bundleId=silktest.InsuranceMobile`.
 - If you want to test an Android app, replace the app with the *appActivity* and the *appPackage*. For example, replace `app=MyApp.apk` with `appActivity=.LoginActivity;appPackage=silktest.insurancemobile`.

For additional information, see [Connection String](#).

Recording Mobile Applications

Once you have established the connection between Silk Test Classic and a mobile device or an emulator, you can record the actions that are performed on the device. To record mobile applications, Silk Test Classic uses a **Recording** window that provides the following functionality:

- Displays the screen of the mobile device or Android emulator which you are testing.
- When you perform an action in the **Recording** window, the same action is performed on the mobile device.
- When you interact with a control on the screen, the **Recording** window preselects the default action.
 - If the default action is a `Click`, and you left-click on the control, the action is executed. You can perform a right-click to show a list of the available actions against the control. You can then select the action that you want to perform and click **OK**.
 - If the default action is not a `Click`, a list of all the available actions against the control displays, and you can select the action that you want to perform or simply accept the preselected action by clicking **OK**.

When you have selected an action from the list, you can type values for the parameters of the selected action into the parameter fields. Silk Test Classic automatically validates the parameters.

- When you pause the recording, you can perform actions in the screen which are not recorded to bring the device into a state from which you want to continue recording.
- When you stop recording, a script is generated with your recorded actions, and you can proceed with replaying the test.

Selecting the Mobile Device for Test Replay

You can define the mobile device that is used for the replay of a test in the following ways:

- If you execute a script from the command line or from a Continuous Integration (CI) server, specify the connection string in the application configuration of the script.
- If you execute a test from Silk Central, specify the mobile device in the **Mobile Device Selection** area of the **Deployment** tab of the execution definition in Silk Central instead of specifying a connection string. For additional information, refer to the [Silk Central Help](#).

You can use the connection string to specify a specific mobile device, or you can filter a subset of the available devices, for example if you have a device pool. The first matching device is used for replay. If not specified otherwise, mobile devices are matched by using the following rules, with declining priority:

- Matching mobile devices connected to the local machine are preferred over mobile devices connected to remote locations.
- If the browser type is specified in the connection string, newer browser versions are preferred over older versions.
- Newer platforms are preferred over older platforms.
- A physical device is preferred to an Emulator or Simulator.
- A device with a device name that is alphabetically later is preferred. For example, a device named "iphone 6" is preferred to a device named "iphone 5".

Example: Connection string for an app on an Android device that is connected to a remote machine

To test the app `MyApp.apk` on an Android device that is connected to a remote machine, the connection string would look like the following:


```
"platformName=Android;deviceName=MotoG3;host=http://10.0.0.1;app=MyApp.apk"
```

Example: Connection string for an app on an iOS Simulator on a Mac

```
"platformName=iOS;platformVersion=9.2;deviceName=iPhone6;host=10.0.0.1;app=MyApp.ipa;isSimulator=true"
```

Connection String for a Mobile Device

The *connection string* specifies which mobile device is used for testing. When performing mobile testing, Silk Test Classic uses the connection string to connect to the mobile device. The connection string is typically part of the application configuration. You can set the connection string when you configure your application under test. To change the connection string, you can use the **Edit Application Configuration** dialog box.

 **Note:** If you execute a test from Silk Central, specify the mobile device in the **Mobile Device Selection** area of the **Deployment** tab of the execution definition in Silk Central instead of specifying a connection string. For additional information, refer to the [Silk Central Help](#).

You can use the connection string to specify a specific mobile device, or you can filter a subset of the available devices, for example if you have a device pool. The first matching device is used for replay. If not specified otherwise, mobile devices are matched by using the following rules, with declining priority:

- Matching mobile devices connected to the local machine are preferred over mobile devices connected to remote locations.
- If the browser type is specified in the connection string, newer browser versions are preferred over older versions.

- Newer platforms are preferred over older platforms.
- A physical device is preferred to an Emulator or Simulator.
- A device with a device name that is alphabetically later is preferred. For example, a device named "iphone 6" is preferred to a device named "iphone 5".

The following components are available for the connection string:

Component	Description
deviceName	The name of the mobile device. When testing on a physical mobile device, the device ID can be used instead. Supports wildcards. Case-insensitive.
platformName	Android or iOS. Required.
deviceId	<i>Optional:</i> The ID of the mobile device. Can be used instead of the device name, when testing on a physical mobile device. Supports wildcards. Case-insensitive.
platformVersion	<i>Optional:</i> The Android or iOS version. Specify the version to test only on mobile devices that have a specific Android or iOS version. Supports wildcards. Case-insensitive.
browserVersion	<i>Optional:</i> Can be used in combination with the browser type to test only on the specified browser version. Supports wildcards. Case-insensitive.
host	<i>Optional:</i> If not set, any specified remote location can be used as the host. Supports wildcards. Case-insensitive.
<ul style="list-style-type: none"> • app • appActivity • appPackage 	Required for testing native mobile applications on Android. Either the full path to the app, or a combination of <i>appActivity</i> and <i>appPackage</i> . For example <code>app=MyApp.apk</code> or <code>appActivity=.LoginActivity;appPackage=silktest.insurancemobile</code>
<ul style="list-style-type: none"> • app • bundleId 	Required for testing native mobile applications on iOS. Either the full path to the app or the <i>bundleId</i> . For example <code>app=MyApp.ipa</code> or <code>bundleId=silktest.InsuranceMobile</code>
noReset	<i>Optional:</i> Can be set when testing native mobile applications. Is only valid if the <i>app</i> is specified. True if the app should not be reinstalled before testing. False if the app should be reinstalled before testing. The default value is False.
isSimulator	<i>Optional:</i> Used to specify that the test should only be executed on an iOS Simulator. The device name can be used instead.
isPhysicalDevice	<i>Optional:</i> Used to specify that the test should only be executed on a physical device. The device name can be used instead.

When using a pool of devices and to find out which device is actually used for testing, you can use the return value of the `GenerateConnectionString` method of `MobileDevice` class.

Testing a mobile web application on a mobile device or on an Android Emulator

When testing a mobile web application on a mobile device or on an Android Emulator, the connection string consists of the following parts:

1. The mobile device name, for example MotoG3, or the device ID, for example 11111111.



Note: If the device name is unique, Micro Focus recommends to use the device name in the connection string, because the device ID is less readable.

2. The platform name.
3. The browser version. This can only be used in combination with setting the browser type
4. The IP address or the host name of a specific remote machine, for example 10.0.0.1. You can also use the name of a remote location that is specified in the **Edit Remote Locations** dialog box as the host name, for example *MyRemoteLocation*. When using the remote location name, you can also use wildcards. To test an Android device that is connected to the local machine, specify the IP address or the host name of the local machine.

Example: Connection string for any available Android device

```
"platformName=Android"
```

Example: Connection string for a browser on an Android device that is connected to the local machine

To test a mobile browser on an Android device that is connected to the local machine, the connection string should look similar to the following:

```
"deviceName=MotoG3;platformName=Android;host=localhost"
```

or

```
"platformName=Android;deviceId=11111111;host=localhost"
```

Example: Connection string for a browser on an Android device that is connected to a remote machine

To test a mobile browser on a remote Android device, the connection string should look similar to the following:

```
"deviceName=MotoG3;platformName=Android;host=10.0.0.1"
```

```
"deviceName=MotoG3;platformName=Android;host=MyRemoteLocation*"
```

Example: Connection string for a browser on an iOS device that is connected to a Mac

To test a mobile browser on a remote iOS device, the connection string would look like the following:

```
"deviceName=myiPhone6;platformName=iOS;host=10.0.0.1"
```

Testing a native mobile application on a mobile device or on an Android Emulator

When testing a native mobile application on a mobile device or on an Android Emulator, the connection string consists of the following parts:

1. The mobile device name, for example MotoG3, or the device ID, for example 11111111.



Note: If the device name is unique, Micro Focus recommends to use the device name in the connection string, because the device ID is less readable.

2. The platform name.
3. The IP address or the host name of a specific remote machine, for example 10.0.0.1. You can also use the name of a remote location that is specified in the **Edit Remote Locations** dialog box as the host name, for example *MyRemoteLocation*. When using the remote location name, you can also use wildcards. To test an Android device that is connected to the local machine, specify the IP address or the host name of the local machine.
4. The name of the file of the app that you want to test, or the URL of the file, if the file is located on a web server. For example `C:/MyApp.apk` or `MyApp.ipa`.
 - Android apps are always .apk files.
 - iOS apps on a real device are always .ipa files.
 - iOS apps on a Simulator are either a zipped file or a directory with the name *app*.

Example: Connection string for an app on an Android device that is connected to a remote machine

To test the app `MyApp.apk` on an Android device that is connected to a remote machine, the connection string would look like the following:

```
"platformName=Android;deviceName=MotoG3;host=http://10.0.0.1;app=MyApp.apk"
```

Example: Connection string for an app on an iOS device that is connected to a Mac

To test the app `MyApp.ipa` on an iOS device that is connected to a remote machine, the connection string would look like the following:

```
"platformName=iOS;deviceName=MyiPhone;host=http://10.0.0.1;app=MyApp.ipa"
```

Testing a mobile web application on an iOS Simulator

When testing a mobile web application on an iOS Simulator, the connection string consists of the following parts:

1. The platform name, which is `iOS`.
2. The platform version, for example `9.2`.
3. The mobile device name, for example `iPhone6`.
4. The IP address or the host name of the Mac, on which the iOS Simulator is running.

Example: Connection string for a browser on an iOS Simulator on a Mac

```
"platformName=iOS;platformVersion=9.2;deviceName=iPhone6;host=10.0.0.1;isSimulator=true"
```

Testing a native mobile application on an iOS Simulator

When testing a native mobile application on an iOS Simulator on a Mac, the connection string consists of the following parts:

1. The platform name, which is `iOS`.
2. The platform version, for example `9.2`.
3. The mobile device name, for example `iPhone6`.
4. The IP address or the host name of the remote machine, for example `10.0.0.1`.
5. The name of the app that you want to test, for example `MyApp.ipa`.

Example: Connection string for an app on an iOS Simulator on a Mac

```
"platformName=iOS;platformVersion=9.2;deviceName=iPhone6;host=10.0.0.1;app=MyApp.ipa;isSimulator=true"
```

Interacting with a Mobile Device

To interact with a mobile device and to perform an action like a swipe in the application under test:

1. In the **Recording** window, click **Show Mobile Device Actions**. All the actions that you can perform against the mobile device are listed.

2. Select the action that you want to perform from the list.
3. To record a swipe on an Android device or emulator, move the mouse while holding down the left mouse button.
4. Continue with the recording of your test.

Releasing a Mobile Device

When recording or playing back a test against a mobile device, the Open Agent instance takes ownership of the device. By doing so, the Open Agent is preventing other Silk Test users from using the device. To enable other Silk Test users to use the device after you have finished recording or replaying tests on the device, Silk Test automatically releases the device when the Silk Test client is closed, when an unattended test process finishes, or when the Open Agent is closed. You can also manually release the device.



Note: Releasing a mobile device will close the application under test (AUT) on the mobile device.

Releasing a Mobile Device After Recording

Release a mobile device after recording to enable other Silk Test users to test on the device.

To release a mobile device after you have finished recording, perform one of the following actions:

- Stop the Open Agent from the System Tray.
- Close the current project.
- Close Silk Test Classic. The device is only released by this action when parallel testing is enabled.



Note: Releasing a mobile device will close the application under test (AUT) on the mobile device.

Releasing a Mobile Device After Replay

Release a mobile device after replay to enable other Silk Test users to test on the device.

To manually release a mobile device after replaying is complete, you can also perform one of the following:

- If you have tested a mobile web application, use the `Close` method of the `BrowserApplication` class. For additional information on these methods, refer to the API documentation.
`WebBrowser.Close()`
- If you have tested a mobile native application, use the `CloseApp` method of the `MobileDevice` class.

For example, type the following:

```
device.CloseApp()
```

- Add the `Agent.DetachAll()` statement to the test script. For example, to release a device when all tests within a test script have completed, type the following into the test script:

```
[ - ] ScriptExit(BOOLEAN bException)
[ ] //Release the mobile device
[ ] Agent.DetachAll()
[ ]
[ ] DefaultScriptExit(bException)
```

A mobile device is automatically released if one of the following conditions is met:

- The Open Agent is closed.
- The test process stops during unattended testing. The device is only released by this action when parallel testing is enabled.
- Silk Test Classic is closed. The device is only released by this action when parallel testing is enabled.



Note: Releasing a mobile device will close the application under test (AUT) on the mobile device.

Troubleshooting when Testing Mobile Applications

Why does the Select Application dialog box not display my mobile browsers?

Silk Test Classic might not recognize a mobile device or emulator for one of the following reasons:

Reason	Solution
The emulator is not running.	Start the emulator.
The Android Debug Bridge (adb) does not recognize the mobile device.	To check if the mobile device is recognized by adb: <ol style="list-style-type: none">1. Navigate to the Android Debug Bridge (adb) in the Android SDK installation folder. If the Android SDK is not installed, navigate to <code>C:\Program Files (x86)\Silk\SilkTest\ng\agent\plugins\com.microfocus.silktest.adb_17.5.0.<build number>\bin</code> to use the adb that is installed with Silk Test Classic.2. Hold Shift and right-click into the File Explorer window.3. Select Open command window here.4. In the command window, type <code>adb devices</code> to get a list of all attached devices.5. If your device is not listed, check if USB-debugging is enabled on the device and if the appropriate USB driver is installed.6. If you get the error <code>adb server is out of date</code>, an additional adb server might be running. For additional information, see <i>What can I do if the connection between the Open Agent and my device is unstable?</i>.
The version of the operating system of the device is not supported by Silk Test Classic.	For information on the supported mobile operating system versions, refer to the Release Notes .
The USB driver for the device is not installed on the local machine.	Install the USB driver for the device on the local machine. For additional information, see <i>Installing a USB Driver</i> .
USB-debugging is not enabled on the device.	Enable USB-debugging on the device. For additional information, see <i>Enabling USB-Debugging</i> .

Why does Silk Test Classic search for a URL in Chrome for Android instead of navigating to the URL?

Chrome for Android might in some cases interpret typing an URL into the address bar as a search. As a workaround you can manually add a command to your script to navigate to the URL.

What do I do if the adb server does not start correctly?

When the Android Debug Bridge (adb) server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients. All adb clients use port 5037 to communicate with the adb server. The adb server locates emulator and device instances by scanning odd-numbered ports in the range 5555 to 5585, which is the range used by emulators and devices. Adb does not allow changing those ports. If you encounter a problem while starting adb, check if one of the ports in this range is already in use by another program.

For additional information, see <http://developer.android.com/tools/help/adb.html>.

What can I do if the connection between the Open Agent and my device is unstable?

If you have installed the Android SDK or another tool that uses the Android Debug Bridge (adb), an additional adb server might be running in addition to the one that is used by Silk Test Classic. If the running adb servers have different versions, the connection between the Open Agent and the device might become unstable or even break.

To avoid version mismatch errors, specify the path to the Android SDK directory by setting the environment variable `SILKTEST_ANDROID_HOME`, for example to `C:\Users\\AppData\Local\Android\android-sdk`. If the variable is not set, the adb version that is shipped with Silk Test Classic is used.

Why do I get the error: Failed to allocate memory: 8?

This error displays if you are trying to start up the emulator and the system cannot allocate enough memory. You can try the following:

1. Lower the RAM size in the memory options of the emulator.
2. Lower the RAM size of Intel HAXM. To lower the RAM size, run the `IntelHaxm.exe` again and choose **change**.
3. Open the **Task Manager** and check if there is enough free memory available. If not, try to free up additional memory by closing a few programs.

Why do I get the error "Silk Test cannot start the app that you have specified" during testing on an iOS device?

This error might display for one or more of the following reasons:

Reason	Solution
The iOS device is not in developer mode.	You can enable the developer mode in one of the following two ways: <ul style="list-style-type: none">• Connect the device to a Mac on which Xcode is installed, and start the app that you want to test on the device.• Add your provisioning profiles to the device.<ol style="list-style-type: none">1. Open Xcode.2. Select Window > Devices.3. Right-click on the iOS device.4. Select Show Provisioning Profiles.5. Add your provisioning profiles.
You have recently updated the iOS version of the device.	<ol style="list-style-type: none">1. Open Xcode.2. Select Window > Devices.3. Wait until Xcode has processed the symbol files.
UI automation is not enabled on the iOS device.	<ol style="list-style-type: none">1. Select Settings > Developer.2. Activate Enable UI Automation.
The Web Inspector is not activated on the iOS device, while you are trying to test a mobile web application.	<ol style="list-style-type: none">1. Click Settings > Safari > Advanced.2. Activate the Web Inspector.
The app that you want to test was not built for the iOS version of the iOS device on which you are testing.	Use Xcode to build the app for the iOS version of the device.

Reason	Solution
The Software Update dialog box is currently open on the iOS device.	<p>Close the dialog box and disable automatic software updates:</p> <ol style="list-style-type: none"> 1. Select Settings > App and iTunes Stores > AUTOMATIC DOWNLOADS. 2. Deactivate Updates.

Why does my Android device display only the Back button in the dynamic hardware controls?

If the Android or the Android Emulator is screen-locked when you start testing, the device or Emulator might display only the button **Back** in the dynamic hardware controls.

To solve this issue, stop the Open Agent, restart the device, and change the device settings to no longer lock the screen.

Why does my Android device or emulator no longer display a keyboard?

To support unicode characters, Silk Test Classic replaces the standard keyboard with a custom keyboard. When testing is finished, the original keyboard is restored. If an error occurs during testing, the custom keyboard might still be active and cannot be replaced.

To solve this issue, manually reset the keyboard under **Settings > Language & input > Current Keyboard.**

Why does my device not respond during testing?

If the device, emulator, or Simulator is screen-locked when you start testing, and Silk Test Classic is unable to unlock the screen, the device, emulator, or Simulator might stop responding to any actions.

To solve this issue, stop the Open Agent and change the device settings to no longer lock the screen.

Why can I not install the Information Service on a Mac?

When the **Allow apps downloaded from** setting in the **General** tab of the **Security & Privacy** system preferences pane is set to **Mac App Store and identified developers**, which is the default value, the following error message appears when opening the Information Service setup:
 "SilkTestInformationService<version>.pkg" can't be opened because it is from an unidentified developer.

To solve this issue, use one of the following:

- Right-click the setup file and select **Open**. A warning message will appear, but you will still be able to open the file.
- Set the **Allow apps downloaded from** setting to **Anywhere**.
- After attempting to open the file, navigate to the **General** tab of the **Security & Privacy** system preferences pane and click **Open Anyway**.

Why is the Recording window black when recording an Android app?

Android apps that require a higher level of security, for example apps that handle financial transactions, might have the **FLAG_SECURE** flag set, which prevents Silk Test Classic from capturing the app. Silk Test Classic relies on screenshots or on a video of the Android device during recording and will display a black screen of the device in the **Recording** window, if the Android app that you are testing has this flag set. To test such an app with Silk Test, you have to contact the app development team, and ask them to un-set the **FLAG_SECURE** flag during testing.

Why does Silk Test Classic not show a video when testing on an Android emulator?

If the emulator is using the graphic card of your computer for better rendering, the video capturing of Silk Test Classic might not work. To solve this, emulate the graphics in software:

1. Open the **Android Virtual Device Manager**.
2. Click **Edit** in the **Actions** column of the emulator.
3. Select **Software** from the list in the **Emulated Performance** area of the **Virtual Device Configuration** dialog.

How can I change the installed version of Xcode?

If the version of Xcode that you are using is not supported by Silk Test Classic, for example when you upgrade to the latest version of Xcode, an error message might appear when testing on iOS.

To replace the installed version of Xcode with a supported version, download a supported Xcode version from <https://developer.apple.com/download/more/>, and replace the unsupported version with the downloaded version. For information about the supported Xcode versions, refer to the [Release Notes](#).

What can I do if my Mac runs out of disk space?

Silk Test Classic uses Instruments to automate iOS devices. This tool creates large log files in the `/Library/Caches/com.apple.dt.instruments` directory, which might fill up disk space on the Mac. To solve this issue, Micro Focus recommends regularly deleting these log files, either manually or by using a cronjob. For example, to delete the files each day at the same time, you could do the following:

1. Type `sudo crontab -e` into a Terminal. This opens an editor in which you can edit the crontab for root.
2. Add the following line to the crontab:

```
0 2 1 * * find /Library/Caches/com.apple.dt.instruments -mtime +10 -delete
```
3. Save the crontab.

In this example, all log files that are older than ten days will be deleted each day at 2 AM from the directory.

Limitations for Testing Mobile Web Applications

The support for playing back tests and recording locators on mobile browsers is not as complete as the support for the other supported browsers. The following list lists the known limitations for playing back tests and recording locators on mobile browsers:

- The following classes, interfaces, methods, and properties are currently not supported for mobile web applications:
 - `BrowserApplication` class.
 - `CloseOtherTabs` method
 - `CloseTab` method
 - `ExistsTab` method
 - `GetActiveTab` method
 - `GetSelectedTab` method
 - `GetSelectedTabIndex` method
 - `GetSelectedTabName` method
 - `GetTabCount` method
 - `ImageClick` method
 - `OpenTab` method
 - `SelectTab` method
 - `DomElement` class.

- DomDoubleClick method
- DomMouseMove method
- GetDomAttributeList method
- IKeyable interface.
 - PressKeys method
 - ReleaseKeys method
- Silk Test Classic does not support testing HTML frames and iFrames with Apple Safari on iOS.
- Recording in landscape mode is not supported for emulators that include virtual buttons in the system bar. Such emulators do not correctly detect rotation and render the system bar in landscape mode to the right of the screen, instead of the lower part of the screen. However, you can record against such an emulator in portrait mode.
- Only HTML attributes in the HTML DOM are supported in XPath expressions for mobile applications. Silk Test Classic does not support properties in XPath expressions.
- If you are testing a mobile web application against Chrome for Android, Silk Test Classic does not support zooming and scrolling.
- The following JavaScript alert-handling methods of the `BrowserWindow` class do not work when testing on the Original Android Stock (AOSP) Browser:
 - AcceptAlert method
 - DismissAlert method
 - GetAlertText method
 - IsAlertPresent method
- At any given point in time, each user on a Mac can only test on one iOS device that is connected to the Mac or on one iOS Simulator that is running on the Mac. To test on multiple iOS Simulators on the same Mac, a user session on the Mac is required for each Simulator. For additional information, see [Testing Multiple iOS Simulators on the Same Mac](#).
- Before starting to test a mobile web application, ensure that no browser tab is open.
- While testing a mobile web application, you can only have one browser tab open.
- Silk Test Classic does not support testing mobile web applications that are opened by a native mobile application.

Limitations for Testing Native Mobile Applications

The known limitations for playing back tests and recording locators on native mobile applications (apps) are:

- The following classes, interfaces, methods, and properties are currently not supported for native mobile applications:
 - IKeyable interface.
 - PressKeys method
 - ReleaseKeys method
 - MobileDevice class.
 - When testing a native mobile application on iOS, the `SetLocation` method is only supported for iOS simulators, not for physical iOS devices.
 - When testing a native mobile application on an Android version prior to Android 6.0, you have to enable **Allow mock locations** to use the `SetLocation` method. To do so, open the settings of the Android device or emulator and tap **Developer Options**.
 - When testing a native mobile application on Android 6.0 or later, you have to set the app to **Appium Settings** to use the `SetLocation` method. To do so, open the settings of the Android device or emulator and tap **Developer Options > Select mock location app**. Then choose **Appium Settings**.



Note: The **Appium Settings** entry is only available if you have already executed a test with Appium on the Android device or emulator.

- Recording in landscape mode is not supported for emulators that include virtual buttons in the system bar. Such emulators do not correctly detect rotation and render the system bar in landscape mode to the right of the screen, instead of the lower part of the screen. However, you can record against such an emulator in portrait mode.
- Only HTML attributes in the HTML DOM are supported in XPath expressions for mobile applications. Silk Test Classic does not support properties in XPath expressions.
- At any given point in time, each user on a Mac can only test on one iOS device that is connected to the Mac or on one iOS Simulator that is running on the Mac. To test on multiple iOS devices or iOS Simulators on the same Mac, a user session on the Mac is required for each device and Simulator. For additional information, see [Testing Multiple iOS Simulators on the Same Mac](#).
- Silk Test Classic does not support text recognition when testing native mobile applications on both Android and iOS.

Text recognition includes the following methods:

- `TextCapture`
 - `TextClick`
 - `TextExists`
 - `TextRectangle`
- Silk Test Classic does not support testing native mobile applications with multiple web views.

Clicking on Objects in a Mobile Website

When clicking on an object during the recording and replay of an automated test, a mobile website presents the following challenges in comparison to a desktop website:

- Varying zoom factors and device pixel ratios.
- Varying screen sizes for different mobile devices.
- Varying font and graphic sizes between mobile devices, usually smaller in comparison to a website in a desktop browser.
- Varying pixel size and resolution for different mobile devices.

Silk Test Classic enables you to surpass these challenges and to click the appropriate object on a mobile website.

When recording a test on a mobile device, Silk Test Classic does not record coordinates when recording a `Click`. However, for cross-browser testing, coordinates are allowed during replay. You can also manually add coordinates to a `Click`. Silk Test Classic interprets these coordinates as the HTML coordinates of the object. To click on the appropriate object inside the `BrowserWindow`, during the replay of a test on a mobile device, Silk Test Classic applies the current zoom factor to the HTML coordinates of the object. The device pixel coordinates are the HTML coordinates of the object, multiplied with the current zoom factor.

If the object is not visible in the currently displayed section of the mobile website, Silk Test Classic scrolls to the appropriate location in the website.

Example

The following code shows how you can test a `DomButton` with a fixed size of 100 x 20 px in your HTML page.

```
DomButton domButton = Desktop.Find("locator for the button")
domButton.Click(MouseButton.LEFT, new Point(50, 10))
```

During replay on a different mobile device or with a different zoom factor, the `DomButton` might for example have an actual width of 10px on the device screen. Silk Test Classic clicks in the middle of the element when using the code above,

independent of the current zoom factor, because Silk Test Classic interprets the coordinates as HTML coordinates and applies the current zoom factor.

Using Existing Mobile Web Tests

Silk Test 17.0 or later uses a different approach to mobile web testing than previous versions of Silk Test. This change might result in your old mobile web tests no longer working on Silk Test 17.0 or later. This topic describes some of the changes that were introduced with Silk Test 17.0 and provides guidance on changing existing mobile web tests with Silk Test 17.0 or later.

The following changes for mobile web testing were introduced with Silk Test 17.0:

- With previous versions of Silk Test, you were able to test on iOS devices that were connected by USB to a Windows machine. With Silk Test 17.0 or later, you can only test on iOS devices that are connected to an OSX machine (Mac).
- If you have tested mobile web applications on an Android device with a previous version of Silk Test, you have to manually remove the proxy from the Android device to test a web application with Silk Test 17.0 or later. Silk Test 17.0 or later no longer requires a proxy, and if the proxy is set, the message `Unable to connect to the proxy server` displays on the device.

Testing Rumba Applications

Rumba is the world's premier Windows desktop terminal emulation solution. Silk Test provides built-in support for recording and replaying Rumba.

When testing with Rumba, please consider the following:

- The Rumba version must be compatible to the Silk Test version. Versions of Rumba prior to version 8.1 are not supported.
- All controls that surround the green screen in Rumba are using basic WPF functionality (or Win32).
- The supported Rumba desktop types are:
 - Mainframe Display
 - AS400 Display
 - Unix Display

For a complete list of the record and replay controls available for Rumba testing, see the *Rumba Class Reference*.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Enabling and Disabling Rumba

Rumba is the world's premier Windows desktop terminal emulation solution. Rumba provides connectivity solutions to mainframes, mid-range, UNIX, Linux, and HP servers.

Enabling Support

Before you can record and replay Rumba scripts, you need to enable support:

1. Install Rumba desktop client software version 8.1 or later.
2. Click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Administration > Rumba plugin > Enable Silk Test Rumba plugin** or (in Microsoft Windows 10) **Start > Silk > Enable Silk Test Rumba plugin**.


Disabling Support

Click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Administration > Rumba plugin > Disable Silk Test Rumba plugin** or (in Microsoft Windows 10) **Start > Silk > Disable Silk Test Rumba plugin**.

Locator Attributes for Identifying Rumba Controls

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests. Supported attributes include:

caption	The text that the control displays.
priorlabel	Since input fields on a form normally have a label explaining the purpose of the input, the intention of priorlabel is to identify the text input field, RumbaTextField , by the text of its adjacent label field, RumbaLabel . If no preceding label is found in the same line of the text field, or if the label at the right side is closer to the text field than the left one, a label on the right side of the text field is used.
StartRow	This attribute is not recorded, but you can manually add it to the locator. Use StartRow to identify the text input field, RumbaTextField , that starts at this row.
StartColumn	This attribute is not recorded, but you can manually add it to the locator. Use StartColumn to identify the text input field, RumbaTextField , that starts at this column.
All dynamic locator attributes.	For additional information on dynamic locator attributes, see <i>Dynamic Locator Attributes</i> .

 **Note:** Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards ? and *.

Testing a Unix Display

Unix displays in Rumba are completely text-based, and provide no UI controls except the main **RUMBA screen** control. To replay a test on a Unix display, you can use the `SendKeys` method to send keys to the Unix display. Silk Test Classic does not support recording on a Unix display.


Rumba Class Reference

When you configure a Rumba application, Silk Test Classic automatically provides built-in support for testing standard Rumba controls.

Testing SAP Applications

Silk Test Classic provides built-in support for testing SAP client/server applications based on the Windows-based GUI module.

For information on the supported versions and any eventual known issues, refer to the *Release Notes*.

 **Note:** If you use SAP NetWeaver with Internet Explorer or Mozilla Firefox, Silk Test Classic tests the application using the xBrowser technology domain.

Silk Test Agent Support

When you create a Silk Test Classic SAP project, the Open Agent is assigned as the default Agent.



Note: You must set **Ctrl+Shift** as the shortcut key combination to use to pause recording. To change the default setting, click **Options > Recorder** and then check the **OPT_ALTERNATE_RECORD_BREAK** check box.

Locator Attributes for SAP Controls

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Silk Test Classic supports the following locator attributes for SAP controls:

- *automationId*
- *caption*



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards `?` and `*`.

Dynamically Invoking SAP Methods

Dynamic invoke enables you to directly call methods, retrieve properties, or set properties, on an actual instance of a control in the application under test. You can also call methods and properties that are not available in the Silk Test Classic API for this control. Dynamic invoke is especially useful when you are working with custom controls, where the required functionality for interacting with the control is not exposed through the Silk Test Classic API.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList` method.

For example, to call a method named `SetTitle`, which requires the title to be set as an input parameter of type string, on an actual instance of a control in the application under test, type the following:

```
control.DynamicInvoke("SetTitle", {"my new title"})
```



Note: Typically, most properties are read-only and cannot be set.



Note: Reflection is used in most technology domains to call methods and retrieve properties.

Supported Methods and Properties

The following methods and properties can be called:

- Methods and properties that Silk Test Classic supports for the control.
- All public methods that the SAP automation interface defines
- If the control is a custom control that is derived from a standard control, all methods and properties from the standard control can be called.

Supported Parameter Types

The following parameter types are supported:

- All built-in Silk Test Classic types

Silk Test Classic types includes primitive types (such as boolean, int, string), lists, and other types (such as Point and Rect).

- UI controls

UI controls can be passed or returned as `AnyWin`.

Returned Values

The following values are returned for properties and methods that have a return value:

- The correct value for all built-in Silk Test Classic types. These types are listed in the *Supported Parameter Types* section.
- All methods that have no return value return `NULL`.

Example

A custom calculator control has a `Reset` method and an `Add` method, which performs an addition of two numbers. You can use the following code to call the methods directly from your tests:

```
customControl.Invoke("Reset")
REAL sum = customControl.DynamicInvoke("Add", {1,2})
```


Configuring Automation Security Settings for SAP

Before you launch an SAP application, you must configure the security warning settings. Otherwise, the security warning `A script is trying to attach to the GUI` displays each time a test plays back an SAP application.


1. Click **Start > Control Panel**.
2. Choose **SAP Configuration**. The **SAP GUI Configuration** dialog box opens.
3. In the **Design Selection** tab, uncheck the **Notify When a Script Attaches to a Running SAP GUI** check box.

SAP Class Reference

When you configure a SAP application, Silk Test Classic automatically provides built-in support for testing standard SAP controls.

 **Note:** To enable Silk Test Classic to recognize SAP controls, enable SAP GUI Scripting on the server. For information on enabling SAP GUI scripting, see [Introduction to SAP Gui Scripting](#). If SAP GUI Scripting is not enabled, Silk Test Classic recognizes all controls as Win32 and custom controls.

The classes included in the SAP class reference, along with all included properties and methods, are part of the SAP Automation module that is directly accessible through Silk Test Classic.

 **Note:** The interface, including the underlying algorithms and the behavior of the interface, is not under the control of Silk Test Classic.

Cross-Browser Testing with the Open Agent

This functionality is supported only if you are using the Open Agent.

With Silk Test Classic, you can easily verify the functionality of even the most advanced web application across a variety of browsers, with a single, portable test script. Silk Test Classic provides leading support for effective and maintainable cross-browser testing with modern web technologies.

One of the main challenges in test automation is to create and maintain test cost effectively. As different browsers behave differently, web application validation is hard to carry out productively. Silk Test Classic enables you to focus on writing tests, as it handles the following three areas of cross-browser testing:

- | | |
|---|--|
| Built-in synchronization | This enables you to create scripts that run on all supported browsers, without the need to manually synchronize against asynchronous events, which are typical of highly dynamic web applications such as AJAX, or HTML5. Silk Test Classic supports synchronization modes for HTML or AJAX as well as all major web environments including Apache Flex, Microsoft Silverlight, and HTML5/AJAX. For additional information, see Page Synchronization for xBrowser . |
| Unified object model | Silk Test Classic enables you to create and maintain a test which runs across a wide range of different browsers. A unified object model across all browsers gives you the ability to focus on a single browser when you create or maintain a test. Silk Test Classic ensures that the object you interact with is accessible in the same way on all the other browsers, which saves time and enables you to focus on testing rather than on finding workarounds for different browsers. |
| Out-of-the-box recording of a cross-browser script | Record a script once and replay it in all the other browsers, without any modifications. This significantly reduces the time and effort it takes to create and maintain test scripts. Nothing is simulated - testing is carried out across real browsers, which ensures that the test behaves exactly as it does for your end user. |

With Silk Test Classic, you can test web applications that use:

- Internet Explorer.
- Mozilla Firefox.
- Google Chrome.
- Microsoft Edge.
- Chrome for Android on an Android device.
- Apple Safari, both on a Mac and on an iOS device.
- Embedded browser controls.



Note: You can record tests for web applications using one of the following browsers:

- Internet Explorer.
- Microsoft Edge.
- Mozilla Firefox.
- Google Chrome 50 or later.
- A mobile browser on a mobile device.

You can replay tests or record locators on any other supported browser.



Note: Before you record or playback web applications, disable all browser add-ons that are installed in your system. To disable add-ons in Internet Explorer, click **Tools > Internet Options**, click the **Programs** tab, click **Manage add-ons**, select an add-on and then click **Disable**.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Sample Applications

To access the Silk Test sample web applications, go to:

- <http://demo.borland.com/InsuranceWebExtJS/>
- <http://demo.borland.com/gmopost>
- <http://demo.borland.com/gmoajax>

Test Objects for xBrowser

Silk Test Classic uses the following classes to model a web application:

Class	Description
BrowserApplication	Exposes the main window of a web browser and provides methods for tabbing.
BrowserObject	Represents the base class for all objects that are contained within a BrowserApplication.
BrowserWindow	Provides access to tabs and embedded browser controls and provides methods for navigating to different pages.
DomElement	Exposes the DOM tree of a web application, including frames, and provides access to all DOM attributes. Specialized classes are available for several DOM elements.
DomButton	Represents the top-level container for a web page. It exposes the DOM tree through DomElement.
DomCheckBox	Represents all DOM elements that were specified using the <code><input type='checkbox'></code> tag.
DomForm	Represents all DOM elements that are specified using the <code><form></code> tag.
DomLink	Represents all DOM elements that were specified using the <code><a></code> tag.
DomListBox	Represents all DOM elements that were specified using the <code><select></code> tag.
DomRadioButton	Represents all DOM elements that were specified using the <code><input type='radio'></code> tag.
DomTable	Represents all DOM elements that were specified using the <code><table></code> tag.
DomTableRow	Represents all DOM elements that were specified using the <code><tr></code> tag.
DomTextField	Represents all DOM elements that were specified using one of the following tags: <ul style="list-style-type: none">• <code><input type='text'></code>• <code><input type='password'></code>• <code><input type='file'></code>• <code><textarea></code>

Overview of Test Frames

A test frame is an include file (`.inc`) that serves as a central global repository of information about the application under test. It contains all the data structures that support your test cases and test scripts. Though you do not have to create a test frame, by declaring all the objects in your application, you will find it much easier to understand, modify, and interpret the results of your tests.

When you create a test frame, Silk Test Classic automatically adds the frame file to the **Use files** field of the **Runtime Options** dialog box. This allows Silk Test Classic to use the information in the declarations and recognize the objects in your application when you record and run test cases.

When you enable extensions, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box. For extensions that use the Open Agent, Silk Test Classic names the include file `<technology_type>.inc`. For instance, if you enable extensions for an Apache Flex application, a file named `flex.inc` is added. If you enable extensions for an Internet Explorer browser, Silk Test Classic adds the `explorer.inc` file to the **Runtime Options** dialog box.

A constant called `wStartup` is created when you record the test frame. By assigning the identifier of the login window to `wStartup` and by recording a new `invoke` method, your tests can start the application, enter any required information into the login window, then dismiss the login window.

See *Marking 4Test Code as GUI-Specific* to learn about the ways you modify the test frame when porting your test cases to other GUIs.

Recording the Test Frame for a Web Application (Open Agent)

This functionality is supported only if you are using the Open Agent.

Record a test frame file for a web application to store information about the web application, for example the URL.

1. Click **File > New** from the menu bar.
2. Click **Test Frame** and then click **OK**. The **Select Application** dialog box displays.
3. Select the **Web** tab.
4. Select one of the installed browsers or mobile browsers from the list.
5. Specify the web page to open in the **Enter URL to navigate** text box.

If an instance of the selected browser is already running, you can click **Use URL from running browser** to record against the URL currently displayed in the running browser instance.

6. Click **OK**.

Modifying the Identifiers

Identifiers are arbitrary strings. You use identifiers to identify objects in your scripts. Tags, on the other hand, are not arbitrary and should not be changed except in well-specified ways.

To make your tests easier to understand and maintain, you can change your objects' identifiers to correspond to their meaning in your application. Then when Silk Test Classic records tests, it will use the identifiers that you specify.

Testing Web Applications on Different Browsers

Testing web applications can be challenging because of the large number of browsers and browser versions available. Your web applications must support the browsers that the users of these applications prefer to use. While developing tests for web applications, you might ask your self the following questions:

- How will your test cases handle differences between browsers?
- How can you specify which browser to use for the test case or test script?

Handling differences between browsers

In most cases, your include files (declarations) and scripts apply to any browser. You can run test cases against different browsers by simply changing the default browser and running the test case, even if the pages look a bit different, such as pushbuttons being in different places. Because Silk Test Classic is object-based, the layout is not relevant. All that matters are the objects on the page.

Declarations and scripts might have one or more lines that apply only to particular browsers. In such a case you can use *browser specifiers* to make lines specific to one or more browsers. Browser specifiers are of the built-in data type `BROWSERTYPE`.

Changing the Browser Type When Replaying Tests from the UI

To select a specific browser to replay a test, perform the following actions:

1. Record the test case against the web application.
For a list of the supported browsers for recording, see [Cross-Browser Testing](#).
2. Replay the test to ensure it works as expected.
3. Click **Configure Applications** on the **Basic Workflow** bar.
If you do not see **Configure Applications** on the **Basic Workflow** bar, ensure that the default Agent is set to the Open Agent.
The **Select Application** dialog box opens.
4. Select the **Web** tab.
5. Select the browser on which you want to replay the test.
6. Click **OK**.
If you have selected an existing instance of Google Chrome, Silk Test Classic checks whether the automation support is included. If the automation support is not included, Silk Test Classic restarts Google Chrome.
The **Choose name and folder of the new frame file** page opens. Silk Test Classic configures the recovery system and names the corresponding file `frame1.inc` by default.
7. Navigate to the location in which you want to save the frame file.
8. In the **File name** text box, type the name for the frame file that contains the default base state and recovery system. Then, click **Save**.
9. Copy the window declaration for the additional browser into the original `frame1.inc` file.
`frame1.inc` must not contain two window declarations of the same name, so rename the Internet Explorer declaration to `InternetExplorer` and the new declaration to another name, for example `Firefox`.
10. To ensure that each of the window declarations works with the appropriate browser type, add the `browserstype` property to the locator.
For example, `"//BrowserApplication[@browserstype='Firefox']"`.
11. Determine which browser you want to use and change the `const wDynamicMainWindow = <browserstype>` to use that browser and then replay the test.
For example, you might type `const wDynamicMainWindow = InternetExplorer`.

Setting the Browser when Running Automated Tests

This functionality is supported only if you are using the Open Agent.

This topic describes how you can set the browser when executing automated tests from a command-line interface, for example from a CI server or from Silk Central. For information on how to set the browser for test replay from the UI, see [Changing the Browser Type When Replaying Tests](#).

1. Record the test case.
2. Use the `sBrowserType` property to specify the browser.

The following browsers are supported for the Open Agent:

```
[ ] const BROWSER_TYPE_INTERNET_EXPLORER = "InternetExplorer"  
[ ] const BROWSER_TYPE_FIREFOX          = "Firefox"  
[ ] const BROWSER_TYPE_CHROME           = "GoogleChrome"  
[ ] const BROWSER_TYPE_ANDROID_BROWSER  = "AndroidBrowser"
```

```
[ ] const BROWSER_TYPE_SAFARI           = "Safari"
[ ] const BROWSER_TYPE_EDGE             = "Edge"
```



Tip: For a complete list of the available browser types, see the file `4Test.inc` file in the Silk Test installation directory, which is by default `C:\Program Files (x86)\Silk\SilkTest`.

3. Call the default base state.
4. Replay the test case.

Examples

The following code shows a sample frame file for the sample web application <http://demo.borland.com/InsuranceWebExtJS/>:

```
[ - ] window BrowserApplication WebBrowser
[ ] locator "//BrowserApplication"
[ ]
[ ] // The start URL
[ ] const sUrl = "http://demo.borland.com"
[ ]
[ ] STRING sBrowserType = NULL
[ ] STRING sConnectionString = NULL
[ ] STRING sCmdLine = NULL
[ ]
[ ] const bCloseOtherTabs = TRUE
```

To run a test against a specific browser, set the `sBrowserType` property and call the base state. The following code sample implements a simple test case that opens the application under test in Internet Explorer:

```
[ - ] testcase OpenGMOAjaxAppIE () appstate none
[ ] WebBrowser.sBrowserType = BROWSER_TYPE_INTERNET_EXPLORER
[ ] WebBrowser.sCmdLine = "C:\\Program Files\\Internet Explorer
\\iexplore.exe" // Optional
[ ] DefaultBaseState()
[ ] WebBrowser.BrowserWindow.DemoApplication.Select()
```

The following code sample opens the same application in Mozilla Firefox:

```
[ - ] testcase OpenGMOAjaxAppFF () appstate none
[ ] WebBrowser.sBrowserType = BROWSER_TYPE_FIREFOX
[ ] DefaultBaseState()
[ ] WebBrowser.BrowserWindow.DemoApplication.Select()
```

To run the test against a remote browser, for example Apple Safari on a Mac, you have to additionally specify the connection string:

```
[ - ] testcase OpenGMOAjaxAppSafari () appstate none
[ - ] recording
[ ] WebBrowser.sBrowserType = BROWSER_TYPE_SAFARI
[ ] WebBrowser.sConnectionString =
"host=macminil702.microfocus.com"
[ ] DefaultBaseState()
[ ] WebBrowser.BrowserWindow.DemoApplication.Select()
```

Testing Objects in a Web Page with the Open Agent

This functionality is supported only if you are using the Open Agent.

The *Document Object Model (DOM)* extension uses information in the HTML source to recognize and manipulate objects on a web page. Silk Test Classic uses the DOM extension to test the objects that are included in a web application.

Using the DOM extension provides several advantages:

- During recording, Silk Test Classic displays a rectangle which highlights the active control in the web page.
- The DOM extension is highly accurate, because it gets information directly from the browser. For example, the DOM extension recognizes text size and the actual name of an object.
- The DOM extension is independent of the browser size and text size settings.
- The DOM extension recognizes non-visible objects.
- The DOM extension offers support for borderless tables.
- The DOM extension is consistent with the standard being developed by the W3C.

Testing Columns and Tables

- If you are using the Classic Agent, tables in Web applications are recognized as `HtmlTable` controls. An `HtmlTable` consists of two or more `HtmlColumn` controls.
- If you are using the Open Agent, tables in Web applications are recognized as `DomTable` controls. Rows in a table are recognized as `DomTableRow` controls.

Definition of a Table (Open Agent)

For the Open Agent, a table is a `DomTable`, which is a DOM element that is specified using the `<table>` tag.

Testing Controls

Web applications can contain the same controls as standard applications, including the following:

Control	Classic Agent Class	Open Agent Class
check box	<code>HtmlCheckBox</code>	<code>DomCheckBox</code>
combo box	<code>HtmlComboBox</code>	No corresponding class.
list boxes	<code>HtmlListBox</code>	<code>DomListBox</code>
popup lists	<code>HtmlPopupList</code>	<code>DomListBox</code>
pushbuttons	<code>HtmlPushButton</code>	<code>DomButton</code>
radio lists	<code>HtmlCheckBox</code>	<code>DomCheckBox</code>

All these classes are derived from their respective standard class. For example, `HtmlCheckBox` is derived from `CheckBox`. So all the testing you can do with these controls in standard applications you can also do in Web applications.

Classic Agent Example

The following code gets the list of items in the credit card list in the **Billing Information** page of the sample GMO application:

```
LIST OF STRING lsCards
lsCards = BillingPage.CreditCardList.GetContents ()
ListPrint (lsCards)
```

```
Result:
American Express
MasterCard
Visa
```

Open Agent Example

The following code gets the list of items in the credit card list in the **Billing Information** page of the sample GMO application:

```
LIST OF STRING lsCards
lsCards = WebBrowser.BrowserWindow.CardType.Items
ListPrint(lsCards)
```

```
Result:
American Express
MasterCard
Visa
```

Testing Images

Classic Agent

If you are using the Classic Agent, images in your Web application are objects of type `HtmlImage`. You can verify the appearance of the image by using the **Bitmap** tab in the **Verify Window** dialog box.

If an `HtmlImage` is an image map, which means that the image contains clickable regions, you can use the following methods to test the clickable regions:

- `GetRegionList`
- `MoveToRegion`
- `ClickRegion`

Open Agent

If you are using the Open Agent, you can test images by using the `IMG` locator. For example, the following code sample finds an image and then prints some of the properties of the image:

```
Window img = FindBrowserApplication("/
BrowserApplication").FindBrowserWindow("//BrowserWindow").Find("//
IMG[@title='Imagel.png']")
String src = img.GetProperty("src")
String altText = img.GetProperty("alt")
print(src)
print(altText)
```

Testing Links

- If you are using the Classic Agent, links in your application are objects of type `HtmlLink`.
- If you are using the Open Agent, links in your application are objects of type `DomLink`.

Silk Test Classic provides several methods that let you get their text properties as well as the location to which they jump.

Classic Agent Example

The following code returns the definition for the `HtmlLink` on a sample home page:

```
STRING sJump
sJump = Acme.LetUsKnowLink.GetLocation ()
Print (sJump)
```

```
Result:
mailto:support@acme.com
```

Open Agent Example

The following code returns the definition for the DomLink on the sample home page:

```
STRING sJump
sJump =
WebBrowser.BrowserWindow.LetUsKnowLink.GetProperty("href")
Print(sJump)
```

```
Result:
mailto:support@acme.com
```

Testing Text in Web Applications

Classic Agent

Straight text in a Web application can be in the following classes:

- HtmlHeading
- HtmlText

Silk Test Classic provides methods for getting the text and all its properties, such as color, font, size, and style.

There are also classes for text in Java applets and applications.

Classic Agent Example

For example, the following code gets the copyright text on a sample Web page:

```
STRING sText
sText = Acme.Copyright.GetText ()
Print (sText)
```

```
Result:
Copyright © 2006 Acme Software, Inc. All rights reserved.
```

Open Agent

When you are using the Open Agent, use the `GetText()` method to get text out of every `DomElement` control.

Open Agent Example

For example, the following code gets the text of a DomLink control:

```
Window link = FindBrowserApplication("/BrowserApplication")
                .FindBrowserWindow("/BrowserWindow")
                .FindDomLink("A[@id='story2128000']")
String linkText = link.GetText()
print(linkText)
```

Object Recognition for xBrowser Objects

This functionality is supported only if you are using the Open Agent.

The xBrowser technology domain supports dynamic object recognition.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an `INC` file to create scripts that use dynamic object recognition and window declarations.

Existing test cases that use dynamic object recognition without locator keywords in an `INC` file will continue to be supported. You can replay these tests, but you cannot record new tests with dynamic object recognition without locator keywords in an `INC` file.

Test cases use locator strings to find and identify objects. A typical locator includes a locator name and at least one locator attribute, such as `"//LocatorName[@locatorAttribute='value']"`.

Locator Names With other technology types, such as Java SWT, locator names are created using the class name of the test object. With `xBrowser`, the tag name of the DOM element can also be used as locator name. The following locators describe the same element:

1. Using the tag name: `"//a[@href='http://www.microfocus.com']"`
2. Using the class name: `"//DomLink[@href='http://www.microfocus.com']"`

To optimize replay speed, use tag names rather than class names.

Locator Attributes All DOM attributes can be used as locator string attributes. For example, the element `<button automationid='123'>Click Me</button>` can be identified using the locator `"//button[@automationid='123']"`.

Recording Locators Silk Test Classic uses a built-in locator generator when recording test cases and using the Locator Spy. You can configure the locator generator to improve the results for a specific application.

xBrowser Default BaseState

This functionality is supported only if you are using the Open Agent.

By default, Silk Test Classic uses the dynamic base state for `xBrowser` projects. When you configure the application, the base state is generated to the `frame.inc` file.

- The `wDynamicMainWindow` variable in the first line of the `frame.inc` file tells Silk Test Classic to use the dynamic base state rather than the classic base state.
- The `WebBrowser` window declaration contains the necessary information to launch the browser and navigate to the Web application that you want to test.
- If you do not want to close all other tabs during base state execution, change `bCloseOtherTabs` to `false`.

Locator Attributes for xBrowser controls

This functionality is supported only if you are using the Open Agent.

When a locator is constructed, the attribute type is automatically assigned based on the technology domain that your application uses. The attribute type and value determines how the locator identifies objects within your tests.

Silk Test Classic supports the following locator attributes for `xBrowser` controls:

caption	Supports wildcards <code>?</code> and <code>*</code> .
all DOM attributes	Supports wildcards <code>?</code> and <code>*</code> .



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards `?` and `*`.



Note: Whitespace, which includes spaces, carriage returns, line feeds, and tabs, is handled differently by each browser. As a result, the `textContent` and `innerText` attributes have been normalized. Whitespace is skipped or replaced by a single space if an empty space is followed by another empty space. The matching of such values is normalized also. In Silk Test 14.0 or later, whitespace in texts, which are retrieved through the `textContent` property of an element, is trimmed consistently.

across all supported browsers. For some browser versions, this whitespace handling differs to Silk Test versions prior to Silk Test 13.5. You can re-enable the old behavior by setting the `OPT_COMPATIBILITY` option to a version lower than 13.5.0.

Example

```
<a>abc abc</a>
```

Uses the following locator:

```
//A[@innerText='abc abc']
```

Page Synchronization for xBrowser

This functionality is supported only if you are using the Open Agent.

Synchronization is performed automatically before and after every method call. A method call is not started and does not end until the synchronization criteria is met.



Note: Any property access is not synchronized.

Synchronization Modes

Silk Test Classic includes synchronization modes for HTML and AJAX.

Using the HTML mode ensures that all HTML documents are in an interactive state. With this mode, you can test simple Web pages. If more complex scenarios with Java script are used, it might be necessary to manually script synchronization functions, such as:

- `WaitForObject`
- `WaitForProperty`
- `WaitForDisappearance`
- `WaitForChildDisappearance`

The AJAX mode synchronization waits for the browser to be in a kind of idle state, which is especially useful for AJAX applications or pages that contain AJAX components. Using the AJAX mode eliminates the need to manually script synchronization functions (such as wait for objects to appear or disappear, wait for a specific property value, and so on), which eases the script creation process dramatically. This automatic synchronization is also the base for a successful record and playback approach without manual script adoptions.

Troubleshooting

Because of the true asynchronous nature of AJAX, generally there is no real idle state of the browser. Therefore, in rare situations, Silk Test Classic will not recognize an end of the invoked method call and throws a timeout error after the specified timeout period. In these situations, it is necessary to set the synchronization mode to HTML at least for the problematic call.



Note: Regardless of the page synchronization method that you use, in tests where a Flash object retrieves data from a server and then performs calculations to render the data, you must manually add a synchronization method to your test. Otherwise, Silk Test Classic does not wait for the Flash object to complete its calculations. For example, you might use `Thread.sleep(milliseconds)`.

Some AJAX frameworks or browser applications use special HTTP requests, which are permanently open in order to retrieve asynchronous data from the server. These requests may let the synchronization hang until the specified synchronization timeout expires. To prevent this situation, either use the HTML synchronization mode or specify the URL of the problematic request in the **Synchronization exclude list** setting.

Use a monitoring tool to determine if playback errors occur because of a synchronization issue. For instance, you can use FindBugs, <http://findbugs.sourceforge.net/>, to determine if an AJAX call is affecting playback. Then, add the problematic service to the **Synchronization exclude list**.



Note: If you exclude a URL, synchronization is turned off for each call that targets the URL that you specified. Any synchronization that is needed for that URL must be called manually. For example, you might need to manually add `waitForObject` to a test. To avoid numerous manual calls, exclude URLs for a concrete target, rather than for a top-level URL, if possible.

Configuring Page Synchronization Settings

You can configure page synchronization settings for each individual test or you can set global options that apply to all tests in the **Script Options** dialog box.

To add the URL to the exclusion filter, specify the URL in the **Synchronization exclude list** in the **Script Options** dialog box.

To configure individual settings for tests,

To configure individual settings for tests, record the test and then insert an agent option to override the global replay value.

For example, you might set the Synchronization mode replay setting to HTML and then return the Synchronization mode to AJAX for the remaining portion of the test if necessary.

To configure individual settings within a test, call any of the following:

- `OPT_XBROWSER_SYNC_MODE`
- `OPT_XBROWSER_SYNC_EXCLUDE_URLS`
- `OPT_SYNC_TIMEOUT`

Setting xBrowser Synchronization Options

This functionality is supported only if you are using the Open Agent.

Specify the synchronization and timeout values for web applications. Synchronization is performed before and after every method call. A method call is not started and does not end until the synchronization criteria is met.

1. Click **Options > Agent** and then click the **Synchronization** tab.
2. From the **Synchronization mode** list box, select the synchronization algorithm for the ready state of a web application.

The synchronization algorithm configures the waiting period for the ready state of an invoke call.

Using the HTML mode ensures that all HTML documents are in an interactive state. With this mode, you can test simple Web pages. If more complex scenarios with Java script are used, it might be necessary to manually script synchronization functions.

Using the AJAX mode eliminates the need to manually script synchronization functions (such as wait for objects to appear or disappear, wait for a specific property value, and so on), which eases the script creation process dramatically. This automatic synchronization is also the base for a successful record and replay approach without manual script adoptions.

3. In the **Synchronization timeout** text box, enter the maximum time, in seconds, to wait for an object to be ready.
4. In the **Synchronization exclude list** text box, type the entire URL or a fragment of the URL for any service or Web page that you want to exclude.

Some AJAX frameworks or browser applications use special HTTP requests, which are permanently open in order to retrieve asynchronous data from the server. These requests may let the synchronization hang until the specified synchronization timeout expires. To prevent this situation, either use the HTML synchronization mode or specify the URL of the problematic request in the **Synchronization exclude** list setting.

For example, if your web application uses a widget that displays the server time by polling data from the client, permanent traffic is sent to the server for this widget. To exclude this service from the synchronization, determine what the service URL is and enter it in the exclusion list.

For example, you might type:

```
http://example.com/syncsample/timeService
timeService
UICallbackServiceHandler
```

Separate multiple entries with a comma.

Note: If your application uses only one service, and you want to disable that service for testing, you must use the HTML synchronization mode rather than adding the service URL to the exclusion list.

5. Click **OK**.

You can now record or manually create a test that uses ignores browser attributes and uses the type of page input that you specified.

Configuring the Locator Generator for xBrowser

This functionality is supported only if you are using the Open Agent.

The Open Agent includes a sophisticated locator generator mechanism that guarantees locators are unique at the time of recording and are easy to maintain. Depending on your application and the frameworks that you use, you might want to modify the default settings to achieve the best results.

A well-defined locator relies on attributes that change infrequently and therefore requires less maintenance. Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index might change when another object is added.

To achieve optimal results, add a custom automation ID to the elements that you want to interact with in your test. In Web applications, you can add an attribute to the element that you want to interact with, such as `<div myAutomationId="my unique element name" />`. This approach can eliminate the maintenance associated with locator changes.

1. Click **Options > Recorder** and then click the **Custom Attributes** tab.
2. If you use custom automation IDs, from the **Select a TechDomain** list box, select **xBrowser** and then add the IDs to the list.

The custom attributes list contains attributes that are suitable for locators. If custom attributes are available, the locator generator uses these attributes before any other attribute. The order of the list also represents the priority in which the attributes are used by the locator generator. If the attributes that you specify are not available for the objects that you select, Silk Test Classic uses the default attributes for xBrowser.

3. Click the **Browser** tab.
4. In the **Locator attribute name exclude list** grid, type the attribute names to ignore while recording.

For example, use this list to specify attributes that change frequently, such as size, width, height, and style. You can include the wildcards "*" and "?" in the Locator attribute name blacklist.

Separate attribute names with a comma.

5. In the **Locator attribute value exclude list** grid, type the attribute values to ignore while recording.

Some AJAX frameworks generate attribute values that change every time the page is reloaded. Use this list to ignore such values. You can also use wildcards in this list.

Separate attribute values with a comma.

6. Click **OK**.

You can now record or manually create a test case.

Comparing API Playback and Native Playback for xBrowser

This functionality is supported only if you are using the Open Agent.

Silk Test Classic supports API playback and native playback for Web applications. If your application uses a plug-in or AJAX, use native user input. If your application does not use a plug-in or AJAX, we recommend using API playback.

Advantages of native playback include:

- With native playback, the agent emulates user input by moving the mouse pointer over elements and pressing the corresponding elements. As a result, playback works with most applications without any modifications.
- Native playback supports plug-ins, such as Flash and Java applets, and applications that use AJAX, while high-level API recordings do not.

Advantages of API playback include:

- With API playback, the Web page is driven directly by DOM events, such as `onmouseover` or `onclick`.
- Scripts that use API playback do not require that the browser be in the foreground.
- Scripts that use API playback do not need to scroll an element into view before clicking it.
- Generally API scripts are more reliable since high-level user input is insensitive to pop-up windows and user interaction during playback.
- API playback is faster than native playback.

Differences Between API and Native Playback Functions

The `DomElement` class provides different functions for API playback and native playback.

The following table describes which functions use API playback and which use native playback.

	API Playback	Native Playback
Mouse Actions	<code>DomClick</code>	<code>Click</code>
	<code>DomDoubleClick</code>	<code>DoubleClick</code>
	<code>DomMouseMove</code>	<code>MoveMouse</code>
		<code>PressMouse</code>
		<code>ReleaseMouse</code>
Keyboard Actions	not available	<code>TypeKeys</code>
Specialized Functions	<code>Select</code>	not available
	<code>SetText</code>	
	etc.	

Setting Recording Options for xBrowser

This functionality is supported only if you are using the Open Agent.

There are several options that can be used to optimize the recording of Web applications.

1. Click **Options > Recorder**.

2. Check the **Record mouse move actions** box if you are testing a Web page that uses mouse move events. You cannot record mouse move events for child technology domains of the xBrowser technology domain, for example Apache Flex and Swing.
Silk Test Classic will only record mouse move events that cause changes to the hovered element or its parent in order to keep scripts short.
3. You can change the **mouse move delay** if required.
Mouse move actions will only be recorded if the mouse stands still for this time. A shorter delay will result in more unexpected mouse move actions.
4. Click the **Browser** tab.
5. In the **Locator attribute name exclude list** grid, type the attribute names to ignore while recording.
For example, if you do not want to record attributes named **height**, add the **height** attribute name to the grid. Separate attribute names with a comma.
6. In the **Locator attribute value exclude list** grid, type the attribute values to ignore while recording.
For example, if you do not want to record attributes assigned the value of **x-auto**, add **x-auto** to the grid. Separate attribute values with a comma.
7. To record native user input instead of DOM functions, check the **OPT_XBROWSER_RECORD_LOWLEVEL** check box.
For example, to record `Click` instead of `DomClick` and `TypeKeys` instead of `SetText`, check this check box.

If your application uses a plug-in or AJAX, use native user input. If your application does not use a plug-in or AJAX, we recommend using high-level DOM functions, which do not require the browser to be focused or active during playback. As a result, tests that use DOM functions are faster and more reliable.
8. Click the **Custom Attributes** tab.
9. Select **xBrowser** in the **Select a tech domain** list box and add the DOM attributes that you want to use for locators to the text box.
Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index might change when another object is added. If custom attributes are available, the locator generator uses these attributes before any other attribute. The order of the list also represents the priority in which the attributes are used by the locator generator. If the attributes that you specify are not available for the objects that you select, Silk Test Classic uses the default attributes for xBrowser.
10. Click **OK**.
You can now record or manually create a test that uses ignores browser attributes and uses the type of page input that you specified.

Browser Configuration Settings for xBrowser

Several browser settings help to sustain stable test executions. Although Silk Test Classic works without changing any settings, there are several reasons that you might want to change the browser settings.


- | | |
|---|--|
| Increase replay speed | Use <code>about:blank</code> as home page instead of a slowly loading Web page. |
| Avoid unexpected behavior of the browser | <ul style="list-style-type: none"> • Disable pop up windows and warning dialog boxes. • Disable auto-complete features. • Disable password wizards. |
| Prevent malfunction of the browser | Disable unnecessary third-party plugins. |

The following sections describe where these settings are located in the corresponding browser.

Internet Explorer

The browser settings are located at **Tools > Internet Options**. The following table lists options that you might want to adjust.

Tab	Option	Configuration	Comments
General	Home page	Set to <code>about:blank</code> .	Minimize start up time of new tabs.
General	Tabs	<ul style="list-style-type: none">• Disable warning when closing multiple tabs.• Enable to switch to new tabs when they are created.	<ul style="list-style-type: none">• Avoid unexpected dialog boxes.• Links that open new tabs might not replay correctly otherwise.
Privacy	Pop-up blocker	Disable pop up blocker.	Make sure your Web site can open new windows.
Content	AutoComplete	Turn off completely	<ul style="list-style-type: none">• Avoid unexpected dialog boxes.• Avoid unexpected behavior when typing keys.
Programs	Manage add-ons	Only enable add-ons that are absolutely required.	<ul style="list-style-type: none">• Third-party add-ons might contain bugs.• Possibly not compatible to Silk Test Classic.
Advanced	Settings	<ul style="list-style-type: none">• Disable Automatically check for Internet Explorer updates.• Enable Disable script debugging (Internet Explorer).• Enable Disable script debugging (Other).• Disable Enable automatic crash recovery.• Disable Display notification about every script error.• Disable all Warn ... settings	Avoid unexpected dialog boxes.

 **Note:** Recording a Web application in Internet Explorer with a zoom level different to 100% might not work as expected. Before recording actions against a Web application in Internet Explorer, set the zoom level to 100%.


Mozilla Firefox

You do not have to change browser settings for Mozilla Firefox. Silk Test Classic automatically starts Mozilla Firefox with the appropriate command-line parameters.

 **Note:** To avoid unexpected behavior when testing web applications, disable auto updates for Mozilla Firefox. For additional information, see [Stop automatic updates](#).

Google Chrome

You do not have to change browser settings for Google Chrome. Silk Test Classic automatically starts Google Chrome with the appropriate command-line parameters.

 **Note:** To avoid unexpected behavior when testing web applications, disable auto updates for Google Chrome. For additional information, see [Turning Off Auto Updates in Google Chrome](#).

Connection String for a Remote Desktop Browser

The *connection string* specifies the remote desktop browser that is used for testing. When testing a web application in a remote browser, Silk Test Classic uses the connection string to connect to the remote

location. The connection string is typically part of the application configuration. You can set the connection string when you configure the web application that you want to test. To change the connection string, you can use the **Edit Application Configuration** dialog box.

When testing a web application in a remote browser, the connection string includes only the host, which means the IP address or the host name of the remote machine, for example 10.0.0.1. To select the correct browser, Silk Test Classic uses the connection string in combination with the browser type, which you can also specify in the **Edit Application Configuration** dialog box.

The host name is case-insensitive.



Note: Remote desktop browser testing is only supported for Microsoft Edge on a remote Microsoft Windows machine, and for Apple Safari on a remote Mac.

Connection string example

```
"host=10.0.0.1"
```

Testing with Apple Safari on a Mac

This functionality is supported only if you are using the Open Agent.

This section describes how you can enhance your cross-browser test set by testing Apple Safari on Mac machines that are connected to a Windows machine on which Silk Test Classic is installed.

Prerequisites for Testing with Apple Safari on a Mac

This functionality is supported only if you are using the Open Agent.

Before you can test with Apple Safari on a Mac, ensure that the following prerequisites are met:

- The Mac is connected as a remote location to a Windows machine, on which Silk Test Classic is installed. For additional information, see *Editing Remote Locations*.
- The SafariDriver, which is the WebDriver extension for Apple Safari that inverts the traditional client/server relationship and communicates with the WebDriver client using WebSockets, is installed on the Mac.
- Java JDK is installed on the Mac.
- The information service is installed on the Mac. To get the files that are required for the information service, use the Silk Test installer. For additional information, see [Installing the Silk Test Information Service on a Mac](#).
- To run tests on Apple Safari, the user that has installed the information service needs to be logged in on the Mac.



Tip: Micro Focus recommends to set the Mac to automatically log in the correct user during startup. For additional information, see [Set your Mac to automatically log in during startup](#).

Installing the Silk Test Information Service on a Mac



Note: To install the information service on a Mac, you require administrative privileges on the Mac.

This functionality is supported only if you are using the Open Agent.


To create and execute tests against Apple Safari on a Mac, or against mobile applications on an iOS or Android device that is connected to a Mac, install the Silk Test information service (information service) on the Mac, and then use the **Remote Locations** dialog box to connect a Windows machine, on which Silk Test Classic is installed, to the Mac.

To install the information service on a Mac:

1. Ensure that a Java JDK is installed on the Mac.

2. If you want to test mobile applications on an iOS device, ensure that Xcode is installed on the Mac.
3. Access the information service setup file, `SilkTestInformationService<Version>-<Build Number>.pkg`.
 - If you have downloaded the information service setup file while installing Silk Test, open the folder `macOS` in the Silk Test installation directory, for example `C:\Program Files (x86)\Silk\SilkTest`.
 - If you have not downloaded the information service setup file while installing Silk Test, you can download the setup file from [Micro Focus SupportLine](#).
4. Copy the file `SilkTestInformationService<Version>-<Build Number>.pkg` to the Mac.
5. Execute `SilkTestInformationService<Version>-<Build Number>.pkg` to install the information service.
6. Follow the instructions in the installation wizard.
7. When asked for the password, provide the password of the currently signed in Mac user.
8. When Apple Safari opens and a message box asks whether to trust the `SafariDriver`, click **Trust**.

To complete the installation, the installer logs the current Mac user out. To verify that the information service was installed correctly, log in to the Mac and click on the Silk Test icon in the top-right corner of the screen to see the available devices and browsers.

 **Tip:** If the Silk Test icon does not appear, restart the Mac.

Limitations for Testing with Apple Safari

This functionality is supported only if you are using the Open Agent.

The following are the known limitations for testing with Apple Safari on a Mac:

- The following classes, interfaces, methods, and properties are currently not supported when testing web applications with Apple Safari on a Mac:
 - `BrowserApplication` class.
 - `ClearCache` method
 - `CloseOtherTabs` method
 - `CloseTab` method
 - `ExistsTab` method
 - `GetHorizontalScrollbar` method
 - `GetNextCloseWindow` method
 - `GetSelectedTab` method
 - `GetSelectedTabIndex` method
 - `GetSelectedTabName` method
 - `GetTabCount` method
 - `GetVerticalScrollbar` method
 - `IsActive` method
 - `Minimize` method
 - `OpenContextMenu` method
 - `OpenTab` method
 - `Restore` method
 - `SelectTab` method
 - `SetActive` method
 - `WindowState` property
 - `BrowserWindow` class.

- `AcceptAlert` method
- `DismissAlert` method
- `GetAlertText` method
- `IsAlertPresent` method
- `MouseMove` method
- `PressKeys` method
- `PressMouse` method
- `ReleaseKeys` method
- `ReleaseMouse` method
- `MoveableWin` class.
 - `GetFocus` function.
- Silk Test Classic does not support testing Apache Flex.
- Silk Test Classic does not support testing iframes with a JavaScript source on Apple Safari.
- To test secure web applications over HTTPS on Apple Safari, ensure that any required server certificates are trusted.
- Silk Test Classic does not provide native support for Apple Safari. You cannot test internal Apple Safari functionality. For example, in a test, you cannot change the currently displayed web page by adding text to the navigation bar. As a workaround, you can use API calls to navigate between web pages.
- Silk Test Classic does not support JavaScript dialog API functions for Apple Safari. As a workaround, you could patch such functions so that they are ignored. For additional information, see <https://groups.google.com/forum/#!topic/selenium-developer-activity/qsovJw93g9c>.
- Silk Test Classic does not support text recognition with Apple Safari.

Text recognition includes the following methods:

- `TextCapture`
- `TextClick`
- `TextExists`
- `TextRectangle`
- Silk Test Classic does not support tabbing on Apple Safari.
- To test a multi window application, disable the Apple Safari pup-up blocker. To do so, start Apple Safari and navigate to **Safari Preferences > Security > Block pop-up window**.
- Silk Test Classic does not support testing the dialog box for saving a password. To avoid this dialog box, start Apple Safari, navigate to **Safari Preferences > AutoFill**, and check the **User names and passwords** check box.
- Silk Test Classic does not support properties in XPath expressions for Apple Safari. Only attributes are supported in XPath expressions.
- Silk Test Classic does not support testing web applications which include a Content-Security-Policy HTTP header.

Running Multiple Apple Safari Tests at the Same Time

To execute a test on Apple Safari, you require a Mac that is connected to the Windows machine on which Silk Test is installed. If multiple Apple Safari want to execute tests on Apple Safari, these tests can be executed simultaneously on the same Mac.



Note: Each test that is executed against Apple Safari on the Mac opens an individual instance of Apple Safari. Having too many instances of Apple Safari running simultaneously might reduce the performance of the Mac.

Uninstalling the Silk Test Information Service from a Mac

This functionality is supported only if you are using the Open Agent.

To uninstall the Silk Test information service (information service) from a Mac, for example if you no longer want to execute tests against Apple Safari on the Mac:

1. Create a new shell file, for example `uninstallInfoService.sh`.
2. Type the following code into the new file:

```
#!/bin/sh

if launchctl list | grep com.borland.infoservice ; then
    launchctl unload /Library/LaunchAgents/com.borland.infoservice.plist
    echo "unloading Launch Daemon"
fi

if [ -d "/Applications/Silk" ]
then
    sudo rm -rf /Applications/Silk
fi

if [ -f "/Library/LaunchAgents/com.borland.infoservice.plist" ]
then
    sudo rm /Library/LaunchAgents/com.borland.infoservice.plist
fi

if [ -f "/usr/local/bin/ideviceinstaller" ]
then
    sudo rm /usr/local/bin/ideviceinstaller
fi

exit 0
```

3. In the command line, type `chmod +x uninstallInfoService.sh` to make the shell file executable.
4. Execute the shell file from the command line.

Testing with Google Chrome

This functionality is supported only if you are using the Open Agent.

This section describes how you can enhance your cross-browser test set by testing with Google Chrome.

Silk Test Classic supports recording actions and replaying tests on Google Chrome 50 or later, and supports replaying tests and recording locators on Google Chrome versions prior to version 50.

- When starting to test on Google Chrome 50 or later, with no running instance of Google Chrome open, Silk Test Classic starts a new instance of Google Chrome. This new browser uses a temporary profile without add-ons and with an empty cache.
- When starting to test on an instance of Google Chrome 50 or later, which is already running, Silk Test Classic restarts Google Chrome with the same command line arguments that were used when the instance was initially started. This restart is required to enable the Silk Test Classic automation support.
- When testing with Google Chrome 50 or later, the Google Chrome instance is closed when shutting down the Open Agent or when starting to test another application outside Google Chrome.



Tip: If you want to execute an existing test script with Google Chrome 50 or later, Micro Focus recommends that you use a base state or that you add a command to the test script to navigate to the URL.

Example 1

If the running instance of Google Chrome 50 or later was initially started with the command `C:/Program Files (x86)/Google/Chrome/Application/chrome.exe www.borland.com`, Google Chrome opens to `www.borland.com` after the restart.

Example 2

If the running instance of Google Chrome 50 or later was initially started with the command `C:/Program Files (x86)/Google/Chrome/Application/chrome.exe`, Google Chrome opens to *about:blank* after the restart.

Prerequisites for Replaying Tests with Google Chrome

Command-line parameters

When you use Google Chrome to replay a test or to record locators, Google Chrome is started with the following command:

```
%LOCALAPPDATA%\Google\Chrome\Application\chrome.exe
--enable-logging
--log-level=1
--disable-web-security
--disable-hang-monitor
--disable-prompt-on-repost
--dom-automation
--full-memory-crash-report
--no-default-browser-check
--no-first-run
--homepage=about:blank
--disable-web-resources
--disable-preconnect
--enable-logging
--log-level=1
--safebrowsing-disable-auto-update
--test-type=ui
--noerrdialogs
--metrics-recording-only
--allow-file-access-from-files
--disable-tab-closeable-state-watcher
--allow-file-access
--disable-sync
--testing-channel=NamedTestingInterface:st_42
```

When you use the wizard to hook on to an application, these command-line parameters are automatically added to the base state. If an instance of Google Chrome is already running when you start testing, without the appropriate command-line parameters, Silk Test Classic closes Google Chrome and tries to restart the browser with the command-line parameters. If the browser cannot be restarted, an error message displays.



Note: The command-line parameter `disable-web-security` is required when you want to record or replay cross-domain documents.



Note: To test a web application that is stored in the local file system, navigate to the `chrome://extensions` in Google Chrome and check the **Allow access to file URLs** check box for the **Silk Test Chrome Extension**.

Testing Google Chrome Extensions

You can use one of the following two approaches to test a Google Chrome extension (add-on) with Silk Test Classic:

Install the extension as a .crx file when starting Google Chrome

To test a Google Chrome extension that is installed as a .crx file, add the following command line to the base state:

```
chrome.exe --load-extension=C:/myExtension/myExtension.crx
```



Note: You can only install a single extension in Google Chrome as a .crx file. To install multiple extensions in Google Chrome, use a comma separated list of .crx files. For example:

```
chrome.exe --load-extension=C:/myExtension/  
myExtension.crx,C:/myExtension2/myExtension2.crx
```

Add the extension to a profile Add the extension to a Google Chrome user data directory and use that profile for testing. For additional information, see [Testing Google Chrome with User Data Directories](#).

Testing Google Chrome with User Data Directories

All changes that you make in Google Chrome, for example your home page, what toolbars you use, any saved passwords, and your bookmarks, are all stored in a special folder, which is called a user data directory.

With Silk Test Classic, you can test Google Chrome user data directories by specifying the path to the user data directory in the base state of the application under test. The following command line includes the path to the profile:

```
chrome.exe --user-data-dir="C:/Users/MyUser/AppData/Local/Google/Chrome/User  
Data"
```



Note: When Google Chrome is started by Silk Test Classic, an empty user data directory is used. This ensures that the test starts at a clean state.

Limitations for Testing with Google Chrome

The support for playing back tests and recording locators with Google Chrome is not as complete as the support for the other supported browsers. The following list lists the known limitations for playing back tests and recording locators with Google Chrome:

- Silk Test Classic does not support recording a test on multiple Google Chrome windows.
- Silk Test does not support testing child technology domains of the xBrowser domain with Google Chrome. For example Apache Flex or Microsoft Silverlight are not supported with Google Chrome.
- With Google Chrome 49 or prior, Silk Test does not provide native support for Google Chrome. You cannot test internal Google Chrome functionality. For example, in a test, you cannot change the currently displayed Web page by adding text to the navigation bar through Win32. As a workaround, you can use API calls to navigate between Web pages. Silk Test supports handling alerts and similar dialog boxes through the Alerts API.
- Silk Test Classic does not support text recognition with Google Chrome.

Text recognition includes the following methods:

- `TextCapture`
- `TextClick`
- `TextExists`
- `TextRectangle`
- Silk Test Classic does not support the `GetFocus` method of the `MoveableWin` class.
- Silk Test does not recognize opening the **Print** dialog box in Google Chrome by using the Google Chrome menu. To add opening this dialog box in Google Chrome to a test, you have to send **Ctrl+Shift+P** using the `TypeKeys` method. Internet Explorer does not recognize this shortcut, so you have to first record your test in Internet Explorer, and then manually add pressing **Ctrl+Shift+P** to your test.
- When two Google Chrome windows are open at the same time and the second window is detached from the first one, Silk Test does not recognize the elements on the detached Google Chrome window. However, Silk Test recognizes the elements of any Google Chrome window that is opened by the AUT during replay.

- With Google Chrome 49 or prior and when you want to test a web application with Google Chrome and the **Continue running background apps when Google Chrome is closed** check box is checked, Silk Test cannot restart Google Chrome to load the automation support.
- To replay a test with a Google Chrome version prior to 50, you need to perform one of the following:
 - Start Google Chrome and enable the Silk Test Chrome extension.



Note: If by mistake you have disabled the Silk Test Chrome extension, you have to re-install the extension from the [Chrome Web Store](#).

- If enabling the Silk Test Chrome extension is not possible, because you have no access to the Chrome Web Store, remove the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\[Wow6432Node\]Google\Chrome\Extensions\cjkicfagnoafgjgpnpcdfllcnneidjj`:
 1. In the **Start** menu, type `regedit` into the search box and press **Enter**.
 2. In the **Registry Editor**, navigate to `HKEY_LOCAL_MACHINE\SOFTWARE\[Wow6432Node\]Google\Chrome\Extensions`.
 3. Right click `cjkicfagnoafgjgpnpcdfllcnneidjj` and select **Delete**.
- With Google Chrome 49 or prior and when using Internet Explorer to replay a test, you can use the following code to test `executeJavaScript`:

```
// Java code
desktop.<BrowserWindow> find("//BrowserWindow")
  .executeJavaScript("function foo() { alert('Silk Test'); }");
desktop.<BrowserWindow> find("//BrowserWindow")
  .executeJavaScript("foo();");
```

When replaying tests on Google Chrome, the scripts are not executed in the global context (window), but in a closure. Everything is executed within a function. The first `ExecuteJavaScript` call in the previous code sample will not work with Google Chrome, because the function `foo` is only available as long as the `ExecuteJavaScript` call lasts.

To replay the same test on Google Chrome, you can use the following function expression:

```
// Java code
desktop.<BrowserWindow> find("//BrowserWindow")
  .executeJavaScript("window.foo = function() { alert('Silk Test'); }");
desktop.<BrowserWindow> find("//BrowserWindow")
  .executeJavaScript("window.foo();");
```

The previous code samples will work in Silk4J. The code for the other Silk Test clients is similar. For additional information, refer to the documentation of the `ExecuteJavaScript` method in the Help of your Silk Test client.



Note: If the Silk Test Chrome extension symbol is marked red, this indicates an error with the Silk Test Chrome support.


Testing with Mozilla Firefox

This functionality is supported only if you are using the Open Agent.

This section describes how you can enhance your cross-browser test set by testing with Mozilla Firefox.

Silk Test Classic supports recording actions and replaying tests on Mozilla Firefox.

- When starting to test on Mozilla Firefox, with no running instance of Mozilla Firefox open, Silk Test Classic starts a new instance of Mozilla Firefox. This new browser uses a temporary profile without add-ons and with an empty cache.
- When starting to test on an instance of Mozilla Firefox which is already running, Silk Test Classic restarts Mozilla Firefox with the same command line arguments that were used when the instance was initially started. This restart is required to enable the Silk Test Classic automation support.
- The Mozilla Firefox instance is closed when shutting down the Open Agent or when starting to test another application outside Mozilla Firefox.

 **Tip:** If you want to execute an existing test script with Mozilla Firefox, Micro Focus recommends that you use a base state or that you add a command to the test script to navigate to the URL.

Example 1

If the running instance of Mozilla Firefox was initially started with the command `C:\program files\Mozilla/firefox.exe www.borland.com`, Mozilla Firefox opens to *www.borland.com* after the restart.

Example 2

If the running instance of Mozilla Firefox was initially started with the command `C:\program files\Mozilla/firefox.exe`, Mozilla Firefox opens to *about:blank* after the restart.

Testing Mozilla Firefox with Profiles

All changes that you make in Mozilla Firefox, for example your home page, what toolbars you use, any saved passwords, and your bookmarks, are all stored in a special folder, which is called a profile.

With Silk Test Classic, you can test Mozilla Firefox profiles by specifying either the name of the profile or the path to the profile in the base state of the application under test.


Using the name of the profile You can use the name of the profile to specify the profile in a command line argument that you can add to the base state of the web application that you want to test. The following command line includes the name of the profile:

```
firefox.exe -p myProfile
```

You can configure the named profile in the **Profile Manager**. To start the **Profile Manager**, type `firefox.exe -P` into a command window.

Using the path to the profile You can use the path to the profile to specify the profile in a command line argument that you can add to the base state of the web application that you want to test. The following command line includes the path to the profile:

```
firefox.exe -profile C:/<path to profile folder>
```


 **Note:** If Mozilla Firefox is started by Silk Test Classic, an empty profile is used. This ensures that the test starts at a clean state. If the user manually starts Mozilla Firefox, the default profile is used.

Testing Mozilla Firefox Extensions

You can use one of the following two approaches to test a Mozilla Firefox extension (add-on) with Silk Test Classic:

Install the extension as a .xpi file when starting Mozilla Firefox To test a Mozilla Firefox extension that is installed as a .xpi file, add the following command line to the base state:

```
firefox.exe C:/myExtension/myExtension.xpi
```

 **Note:** When starting Mozilla Firefox from the base state, the confirmation dialog that is by default shown by Mozilla Firefox when installing the extension is suppressed.

Testing a Mozilla Firefox extension that is installed as a .xpi file with Silk Test Classic has the following limitations:

- The .xpi file must exist in the local file system.

- You can only install a single extension in Mozilla Firefox as a .xpi file. To install multiple extensions in Mozilla Firefox, add the extensions to a Mozilla Firefox profile.

Add the extension to a Mozilla Firefox profile Add the extension to a Mozilla Firefox profile and use that profile for testing. For additional information, see [Testing Mozilla Firefox with Profiles](#).

Limitations for Testing with Mozilla Firefox

The following limitations are known when testing web applications with Silk Test Classic on Mozilla Firefox:

- Silk Test Classic does not support Mozilla Firefox 47. However, Silk Test Classic supports Mozilla Firefox 47.0.1
- Silk Test Classic does not support recording a test on multiple Mozilla Firefox windows.
- Silk Test Classic does not support testing modal browser windows, which are windows that can be displayed with the `window.showmodalDialog` command. These modal browser windows have been officially deprecated, and are disabled with Google Chrome 37 or later, while they are planned to no longer be supported in future versions of Mozilla Firefox. You can work around this issue by using low-level actions, for example native clicks with coordinates to click on an object or typekeys to fill out text fields.
- Silk Test Classic does not support testing Silverlight with Mozilla Firefox.
- Silk Test Classic does not support testing some browser dialogs, for example the **About** dialog, with Mozilla Firefox.
- Silk Test Classic does not support testing `about:*` pages with Mozilla Firefox.
- Silk Test Classic does not support text recognition with Mozilla Firefox.

Text recognition includes the following methods:

- `TextCapture`
- `TextClick`
- `TextExists`
- `TextRectangle`
- Silk Test Classic does not support recording locators on JavaScript alert boxes with Mozilla Firefox. You can work around this issue by manually scripting the locators. For example, to click on the button **OK**, which has the locator `//PushButton[@caption='OK']`, type the following into your script:


```
WebBrowser.Dialog.FindPushButton("//PushButton[@caption='OK']").Click()
```
- Silk Test Classic does not support Java applets for Mozilla Firefox 48 or later. Silk Test Classic supports Java applets for Mozilla Firefox 47.0.1 or prior with the following limitations:
 - Silk Test Classic does not support the locator `//AppletContainer`.
 - When the applet opens a modal dialog, a locator similar to `//BrowserApplication//BrowserWindow/JDialog[@caption='Information']//JButton[@caption='OK']` might not work. You can use a locator like `//JDialog[@caption='Information']//JButton[@caption='OK']` instead.
- Silk Test Classic does not support properties in XPath expressions for Mozilla Firefox. Only attributes are supported in XPath expressions.
- Silk Test Classic does not support the `GetFocus` method of the `MoveableWin` class.
- Silk Test Classic does not support testing applications that are based on Apache Flex.
- With Mozilla Firefox 48 or later, the `PressKey` method and the `ReleaseKey` method are not supported.
- With Mozilla Firefox 48 or later, the `MouseMove` method of the `BrowserWindow` class is not supported. Silk Test uses the `DomMouseMove` method instead.
- With Mozilla Firefox 48 or later, native playback for the following is not supported:

- The `Click` method with coordinates. Native playback for this method is simulated with JavaScript. Whenever possible, Micro Focus recommends using `Click` without coordinates.
- Double-click.
- Right and middle mouse button click.
- With Mozilla Firefox 48 or later, the following limitations apply to alert windows:
 - The `AcceptAlert` method with a `text` parameter is not supported.
 - The `DomClick` method is not supported on controls that open an alert.

Testing with Microsoft Edge

This functionality is supported only if you are using the Open Agent.

This section describes how you can enhance your cross-browser test set by testing with Microsoft Edge.

Limitations for Testing with Microsoft Edge

This functionality is supported only if you are using the Open Agent.

The following are the known limitations for testing with Microsoft Edge:

- The following classes, interfaces, methods, and properties are currently not supported when testing web applications on Microsoft Edge:
 - `BrowserApplication` class.
 - `ClearCache` method
 - `CloseOtherTabs` method
 - `CloseTab` method
 - `ExistsTab` method
 - `GetHorizontalScrollbar` method
 - `GetNextCloseWindow` method
 - `GetSelectedTab` method
 - `GetSelectedTabIndex` method
 - `GetSelectedTabName` method
 - `GetTabCount` method
 - `GetVerticalScrollbar` method
 - `IsActive` method
 - `Minimize` method
 - `OpenContextMenu` method
 - `OpenTab` method
 - `Restore` method
 - `SelectTab` method
 - `SetActive` method
 - `WindowState` method
 - The following methods of the `BrowserWindow` class are not supported for Microsoft Edge versions prior to build 38.14393, the Microsoft Edge version for Microsoft Windows 10 Anniversary Update.
 - `PressKeys` method
 - `ReleaseKeys` method
- Silk Test Classic does not automatically bring the browser into the foreground when recording actions against Microsoft Edge.
- When testing with Microsoft Edge, the rectangle for the `BrowserApplication` is not absolute.
- Silk Test Classic does not support testing Apache Flex.

- Silk Test Classic does not support testing iFrames on Microsoft Edge.
- Silk Test Classic does not support recording keyword-driven tests on Microsoft Edge.
- Silk Test Classic does not provide native support for Microsoft Edge. You cannot test internal Microsoft Edge functionality. For example, in a test, you cannot change the currently displayed web page by adding text to the navigation bar. As a workaround, you can use API calls to navigate between web pages.
- Silk Test Classic does not support handling alerts and similar dialog boxes for Microsoft Edge.
- Image clicks are only supported for Microsoft Edge Threshold 2 (build 25.10586) or later. If you are testing a web application on a prior version of Microsoft Edge, you can only use image verifications.
- Silk Test Classic does not support text recognition with Microsoft Edge.

Text recognition includes the following methods:

- `TextCapture`
- `TextClick`
- `TextExists`
- `TextRectangle`
- Silk Test Classic does not support tabbing on Microsoft Edge. Tabs are recognized as windows.
- When testing web applications on Microsoft Edge, Silk Test Classic cannot locate meta-tags which include the `http-equiv` attribute. For example, Silk Test Classic cannot locate the following meta-tag:


```
<meta http-equiv="content-type" content="text/html; charset=utf-8">
```
- With Microsoft Edge, Silk Test Classic does not support directly reading the `currentStyle` attribute of a DOM element. You can use the `GetCssStyle` method of the `DomElement` class to retrieve the computed CSS style with the specified style name.
- When starting the interaction with a web application on Microsoft Edge, Silk Test Classic closes any open instance of Microsoft Edge and starts a new browser. This new browser uses a temporary profile without add-ons and with an empty cache. This instance of Microsoft Edge is closed when shutting down the Open Agent or when starting to test another application outside Microsoft Edge.
- With Microsoft Edge, Silk Test Classic does not recognize the `textContent` attribute while recording actions or locators. However, you can use the `textContent` attribute in object maps and when replaying a test on Microsoft Edge.
- Silk Test Classic does not support properties in XPath expressions for Microsoft Edge. Only attributes are supported in XPath expressions.
- Silk Test Classic does not support the `GetFocus` method of the `MoveableWin` class.
- Silk Test Classic does not support testing web applications which include a Content-Security-Policy HTTP header.

Testing Microsoft Edge on a Remote Machine

To test Microsoft Edge on a remote Microsoft Windows 10 machine, Silk Test Classic needs to be installed on the remote machine.

1. On the local machine, from which you want to test Microsoft Edge, add the remote Microsoft Windows 10 machine as a remote location.

For additional information, see [Editing Remote Locations](#).

2. If the Silk Test information service is already running with administrator privileges on the remote Microsoft Windows 10 machine, disable the Silk Test information service.
 - a) Sign in to the Microsoft Windows 10 machine as an administrator.
 - b) Open the **Control Panel** (icons view).
 - c) Click **Administrative Tools**.
 - d) Double-click **Services**.
 - e) Double click on the Silk Test information service.
 - f) If the service shows a status of running, click **Stop** and wait until the service status shows as stopped.

- g) Change the **Startup type** to **Disabled**.
 - h) Click **OK**.
3. Ensure that the Silk Test information service is running in a user session, and not in the services session. The Silk Test information service requires UI interaction to be enabled. The services session, which is session 0, does not enable UI interaction.
 4. Start the Silk Test information service with medium integrity level, which means without administrator privileges.
 - a) Open a file explorer and navigate to %OPEN_AGENT_HOME%/InfoService.
For example, C:\Program Files (x86)\Silk\SilkTest\ng\InfoService.
 - b) Double-click `infoservice_start.bat`.
 5. You can now select Microsoft Edge on the remote Microsoft Windows 10 machine in the **Select Application** dialog box.

Manually Creating Tests for Dynamic Popup Menus

You must enable xBrowser extensions and use the Open Agent to create scripts that use dynamic object recognition.

Although the xBrowser extension does not support recording, you can manually create scripts that test dynamic popup menus. When you manually create scripts use the **Record Window Identifiers** dialog box to identify the locator strings for dynamic object recognition. After you determine which event needs to be triggered in order to pop up the menu, trigger it either by using native user input, by moving the mouse over the element or clicking the element, or by triggering the event directly. For example, to trigger the event, type:

```
DomElement.ExecuteJavaScript("currentElement.onmouseover()")
```

Finding Hidden Input Fields

Hidden input fields are HTML fields for which the tag includes `type="hidden"`. To enable a Find to locate hidden input fields, you can use the `OPT_XBROWSER_FIND_HIDDEN_INPUT_FIELDS` option. The default value of the option is `TRUE`.

```
Agent.SetOption(OPT_XBROWSER_FIND_HIDDEN_INPUT_FIELDS, TRUE)
```

Cross-Browser Testing: Frequently Asked Questions

This section includes questions that you might encounter when testing your Web application on various browsers.

How do I Verify the Font Type Used for the Text of an Element?

You can access all attributes of the `currentStyle` attribute of a DOM element by separating the attribute name with a ":".

Internet Explorer 8 or earlier `wDomElement.GetProperty("currentStyle:fontName")`

**All other browsers, for example
Internet Explorer 9 or later and
Mozilla Firefox** `wDomElement.GetProperty("currentStyle:font-name")`

What is the Difference Between `textContent`, `innerText`, and `innerHTML`?

- `textContent` is all text contained by an element and all its children that are for formatting purposes only.


- `innerText` returns all text contained by an element and all its child elements.
- `innerHTML` returns all text, including html tags, that is contained by an element.

Consider the following html code.

```
<div id="mylinks">
  This is my <b>link collection</b>:
  <ul>
    <li><a href="www.borland.com">Bye bye <b>Borland</b> </a></li>
    <li><a href="www.microfocus.com">Welcome to <b>Micro Focus</b></a></li>
  </ul>
</div>
```

The following table details the different properties that return.

Code	Returned Value
<code>browser.DomElement("//div[@id='mylinks']").GetProperty("textContent")</code>	This is my link collection:
<code>browser.DomElement("//div[@id='mylinks']").GetProperty("innerText")</code>	This is my link collection:Bye bye Borland Welcome to Micro Focus
<code>browser.DomElement("//div[@id='mylinks']").GetProperty("innerHTML")</code>	This is my link collection: Bye bye Borland Welcome to Micro Focus

 **Note:** In Silk Test 13.5 or later, whitespace in texts, which are retrieved through the `textContent` property of an element, is trimmed consistently across all supported browsers. For some browser versions, this whitespace handling differs to Silk Test versions prior to Silk Test 13.5. You can re-enable the old behavior by setting the `OPT_COMPATIBILITY` option to a version lower than 13.5.0. For example, to set the option to Silk Test 13.0, type the following into your script:

```
Agent.SetOption(OPT_COMPATIBILITY, "13.0.0")
```

I Configured `innerText` as a Custom Class Attribute, but it Is Not Used in Locators

A maximum length for attributes used in locator strings exists. `InnerText` tends to be lengthy, so it might not be used in the locator. If possible, use `textContent` instead.

What Should I Take Care Of When Creating Cross-Browser Scripts?

When you are creating cross-browser scripts, you might encounter one or more of the following issues:

- Different attribute values. For example, colors in Internet Explorer are returned as "# FF0000" and in Mozilla Firefox as "rgb(255,0,0)".
- Different attribute names. For example, the font size attribute is called "fontSize" in Internet Explorer 8 or earlier and is called "font-size" in all other browsers, for example Internet Explorer 9 or later and Mozilla Firefox.
- Some frameworks may render different DOM trees.

How Can I See Which Browser I Am Currently Using?

The `BrowserApplication` class provides a property `"browserType"` that returns the type of the browser. You can add this property to a locator in order to define which browser it matches.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Examples

To get the browser type, type the following into the locator:

```
browserApplication.GetProperty("browserType")
```

Additionally, the `BrowserWindow` provides a method `GetUserAgent` that returns the user agent string of the current window.

Which Locators are Best Suited for Stable Cross Browser Testing?

The built in locator generator attempts to create stable locators. However, it is difficult to generate quality locators if no information is available. In this case, the locator generator uses hierarchical information and indices, which results in fragile locators that are suitable for direct record and replay but ill-suited for stable, daily execution. Furthermore, with cross browser testing, several AJAX frameworks might render different DOM hierarchies for different browsers.

To avoid this issue, use custom IDs for the UI elements of your application.

Logging Output of My Application Contains Wrong Timestamps

This might be a side effect of the synchronization. To avoid this problem, specify the HTML synchronization mode.

My Test Script Hangs After Navigating to a New Page

This can happen if an AJAX application keeps the browser busy (open connections for Server Push / ActiveX components). Try to set the HTML synchronization mode. Check the *Page Synchronization for xBrowser* topic for other troubleshooting hints.

Recorded an Incorrect Locator

The attributes for the element might change if the mouse hovers over the element. Silk Test Classic tries to track this scenario, but it fails occasionally. Try to identify the affected attributes and configure Silk Test Classic to ignore them.

Rectangles Around Elements in Internet Explorer are Misplaced

- Make sure the zoom factor is set to 100%. Otherwise, the rectangles are not placed correctly.
- Ensure that there is no notification bar displayed above the browser window. Silk Test Classic cannot handle notification bars.

Link.Select Does Not Set the Focus for a Newly Opened Window in Internet Explorer

This is a limitation that can be fixed by changing the Browser Configuration Settings. Set the option to always activate a newly opened window.

DomClick(x, y) Is Not Working Like Click(x, y)

If your application uses the `onclick` event and requires coordinates, the `DomClick` method does not work. Try to use `Click` instead.

FileInputField.DomClick() Will Not Open the Dialog

Try to use `Click` instead.

The Move Mouse Setting Is Turned On but All Moves Are Not Recorded. Why Not?

In order to not pollute the script with a lot of useless `MoveMouse` actions, Silk Test Classic does the following:

- Only records a `MoveMouse` action if the mouse stands still for a specific time.
- Only records `MoveMouse` actions if it observes activity going on after an element was hovered over. In some situations, you might need to add some manual actions to your script.
- Silk Test Classic supports recording mouse moves only for Web applications, Win32 applications, and Windows Forms applications. Silk Test Classic does not support recording mouse moves for child technology domains of the xBrowser technology domain, for example Apache Flex and Swing.

I Need Some Functionality that Is Not Exposed by the xBrowser API. What Can I Do?

You can use `ExecuteJavaScript()` to execute JavaScript code directly in your Web application. This way you can build a workaround for nearly everything.

Why Are the Class and the Style Attributes Not Used in the Locator?

These attributes are on the ignore list because they might change frequently in AJAX applications and therefore result in unstable locators. However, in many situations these attributes can be used to identify objects, so it might make sense to use them in your application.

Dialog is Not Recognized During Replay

When recording a script, Silk Test Classic recognizes some windows as `Dialog`. If you want to use such a script as a cross-browser script, you have to replace `Dialog` with `Window`, because some browsers do not recognize `Dialog`.

For example, the script might include the following line:

```
/BrowserApplication//Dialog//PushButton[@caption='OK']
```

Rewrite the line to enable cross-browser testing to:

```
/BrowserApplication//Window//PushButton[@caption='OK']
```

Why Do I Get an Invalidated-Handle Error?

This topic describes what you can do when Silk Test Classic displays the following error message: `The handle for this object has been invalidated.`

This message indicates that something caused the object on which you called a method, for example `WaitForProperty`, to disappear. For example, if something causes the browser to navigate to a new page, during a method call in a Web application, all objects on the previous page are automatically invalidated.

When testing a Web application, the reason for this problem might be the built-in synchronization. For example, suppose that the application under test includes a shopping cart, and you have added an item to this shopping cart. You are waiting for the next page to be loaded and for the shopping cart to change its status to `contains items`. If the action, which adds the item, returns too soon, the shopping cart on the

first page will be waiting for the status to change while the new page is loaded, causing the shopping cart of the first page to be invalidated. This behavior will result in an invalidated-handle error.

As a workaround, you should wait for an object that is only available on the second page before you verify the status of the shopping cart. As soon as the object is available, you can verify the status of the shopping cart, which is then correctly verified on the second page.

Why Are Clicks Recorded Differently in Internet Explorer 10?

When you record a `Click` on a `DomElement` in Internet Explorer 10 and the `DomElement` is dismissed after the `Click`, then the recording behavior might not be as expected. If another `DomElement` is located beneath the initial `DomElement`, Silk Test records a `Click`, a `MouseMove`, and a `ReleaseMouse`, instead of recording a single `Click`.

A possible workaround for this unexpected recording behavior depends on the application under test. Usually it is sufficient to delete the unnecessary `MouseMove` and `ReleaseMouse` events from the recorded script.

Testing the Insurance Company Sample Web Application

Silk Test Classic provides a sample insurance company Web application, <http://demo.borland.com/InsuranceWebExtJS/>.

To complete a tutorial for how to test the insurance company Web application using Silk Test Classic, complete each of the following steps. Or, in the Help, click the **Contents** tab and then expand **Testing in Your Environment > Testing Web Applications > Using the xBrowser Tech Domain > Testing the Insurance Company Sample Web Application**. Follow the topics sequentially in the **Testing the Insurance Company Sample Web Application** book to test the sample Web application using Silk Test Classic.

To test the sample Web application, follow these steps:

- Create a New Project for the insurance company Web application.
- Configure the insurance company Web application.
- Record a test case for the insurance company Web site.
- Replay the test case for the insurance company Web site.
- Modify the insurance company test case to replay tests in a different browser than Internet Explorer.

Creating a New Project for the Insurance Company Web Application

The type of project that you select determines the default Agent. For Web application projects, the Open Agent is automatically set as the default agent. Silk Test Classic uses the default agent when configuring an application and recording a test case.

1. Click **File > New Project**, or click **Open Project > New Project** on the **Basic workflow** bar. The **Create Project** dialog box opens.
2. Type a project name and a description in the appropriate fields.
3. Click **OK** to save your project in the default location, `C:\Users\<Current user>\Documents\Silk Test Classic Projects`. Silk Test Classic creates a `<Project name>` folder within the specified directory, saves the `projectname.vtp` and `projectname.ini` to this location and copies the extension `.ini` files, which are `appexpex.ini`, `axext.ini`, `domex.ini`, and `javaex.ini`, to the `extend` subdirectory. If you do not want to save your project in the default location, click **Browse** and specify the folder in which you want to save your project.

Silk Test Classic creates your project and displays nodes on the **Files** and **Global** tabs for the files and resources associated with this project.

Configuring the Insurance Company Web Application

When you configure an application, Silk Test Classic automatically creates a base state for the application. An application's base state is the known, stable state that you expect the application to be in before each test begins execution, and the state the application can be returned to after each test has ended execution.

1. Click **Configure Applications** on the **Basic Workflow** bar.

If you do not see **Configure Applications** on the **Basic Workflow** bar, ensure that the default Agent is set to the Open Agent.

The **Select Application** dialog box opens.

2. Select the **Web** tab.
3. Select **Internet Explorer**.
4. In the **Enter URL to navigate** text box, type the URL of the web application that you want to test.
For example, type <http://demo.borland.com/gmoajax>.
5. Click **OK**.

The **Choose name and folder of the new frame file** page opens. Silk Test Classic configures the recovery system and names the corresponding file `frame.inc` by default.

6. Navigate to the location in which you want to save the frame file.
7. In the **File name** field, type the name for the frame file that contains the default base state and recovery system. Then, click **Save**.

Silk Test Classic automatically creates a base state for the application. By default, Silk Test Classic lists the caption of the main window of the application as the locator for the base state. When you configure an application, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box. For instance, if you configure an application that uses one of the supported browsers, Silk Test Classic adds the `xBrowser.inc` file to the **Runtime Options** dialog box.

Silk Test Classic opens the Web page. Record the test case whenever you are ready.

Recording a Test Case for the Insurance Company Web Site

1. Click **Record Testcase** on the **Basic Workflow** bar. The **Record Testcase** dialog box opens.
2. Type the name of your test case in the **Testcase name** field.

For example, type `ZipTest`.

Test case names are not case sensitive; they can be any length and consist of any combination of alphabetic characters, numerals, and underscore characters.

3. From the **Application State** list box, select **DefaultBaseState** to have the built-in recovery system restore the default base state before the test case begins executing. If you choose **DefaultBaseState** as the application state, the test case is recorded in the script file as: `testcase testcase_name ()`.
4. If you do not want Silk Test Classic to display the status window during playback when driving the application to the specified base state, uncheck the **Show AppState status window** check box.

Typically, you check this check box. However, in some circumstances it is necessary to hide the status window. For instance, the status bar might obscure a critical control in the application you are testing.

5. Click **Start Recording**. Silk Test Classic:
 - Closes the **Record Testcase** dialog box.
 - Starts your application, if it is not already running
 - Removes the editor window from the display.
 - Displays the **Recording** window.
 - Waits for you to take further action.

6. In the insurance company web site, perform the following steps:
 - a) From the **Select a Service or login** list box, select **Auto Quote**. The **Automobile Instant Quote** page opens.
 - b) Type a zip code and email address in the appropriate fields, click an automobile type, and then click **Next**.
 - c) Specify an age, click a gender and driving record type, and then click **Next**.
 - d) Specify a year, make, and model, click the financial info type, and then click **Next**. A summary of the information you specified displays.
 - e) Point to the Zip Code that you specified and press **Ctrl+Alt** to add a verification to the script. You can add a verification for any of the information that displays. The **Verify Properties** dialog box opens.
 - f) Check the **textContents** check box and then click **OK**. A verification action is added to the script for the zip code text.An action that corresponds with each step is recorded.
7. To review what you have recorded, click **Stop** in the **Recording** window.

Silk Test Classic displays the **Record Testcase** dialog box, which contains the code that has been recorded for you.

8. Click **Paste to Editor**.
9. Click **File > Save**.

Replay the test to ensure that it works as expected. You can modify the test to make changes if necessary.

Replaying a Test Case for the Insurance Company Web Site

Replay a test to ensure that it works as expected.

1. Make sure that the test case you want to run is in the active window.
2. Click **Run Testcase** on the **Basic Workflow** bar.

Silk Test Classic displays the **Run Testcase** dialog box, which lists all the test cases contained in the current script.

3. Select the ZipTest test case.
4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

- BaseStateExecutionFinished
- Connecting
- Verify
- Exists
- Is
- Get
- Set
- Print
- ForceActiveXEnum
- Wait
- Sleep

5. Click **Run**.

Silk Test Classic runs the test case and generates a results file. The results file describes whether the test passed or failed, and provides summary information.

xBrowser Classes

This section lists the classes that are used for the xBrowser technology domain.

Testing Windows API-Based Applications

This section describes how Silk Test Classic provides built-in support for testing Microsoft Windows API-based applications.

Overview of Windows API-Based Application Support

Silk Test Classic provides built-in support for testing Microsoft Windows API-based applications. Several objects exist in Microsoft applications that Silk Test Classic can better recognize if you enable Accessibility. For example, without enabling Accessibility Silk Test Classic records only basic information about the menu bar in Microsoft Word and the tabs that display in Internet Explorer 7.0. However, with Accessibility enabled, Silk Test Classic fully recognizes those objects. You can also improve Silk Test Classic object recognition by defining a new window, if necessary.

You can test Windows API-based applications using the Classic or Open Agent.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Object Recognition

Windows API-based applications support hierarchical object recognition and dynamic object recognition. You can create tests for both dynamic and hierarchical object recognition in your test environment. Use the method best suited to meet your test requirements.

When you record a test case with the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations.

Existing test cases that use dynamic object recognition without locator keywords in an INC file will continue to be supported. You can replay these tests, but you cannot record new tests with dynamic object recognition without locator keywords in an INC file.

To test Windows API-based applications using hierarchical object recognition, record a test for the application that you want to test. Then, replay the tests at your convenience.

Supported Controls

For a complete list of the record and replay controls available for Windows-based testing for each Agent type, view the `WIN32.inc` and `winclass.inc` file. To access the `WIN32.inc` file, which is used with the Open Agent, navigate to the `<SilkTest directory>\extend\WIN32` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\extend\WIN32\WIN32.inc`. To access the `winclass.inc` file, which is used with the Classic Agent, navigate to the `<SilkTest directory>\` directory. By default, this file is located in `C:\Program Files\Silk\SilkTest\winclass.inc`.

Locator Attributes for Windows API-Based Applications

Silk Test Classic supports the following locator attributes for the controls of Windows API-based client/server applications:

- `caption`.
- `windowid`.

- `priorlabel`. For controls that do not have a caption, `priorlabel` is used as the caption automatically. For controls with a caption, it may be easier to use the caption.



Note: Attribute names are case sensitive. Attribute values are by default case insensitive, but you can change the default setting like any other option. The locator attributes support the wildcards `?` and `*`.

Suppressing Controls (Classic Agent)

This functionality is supported only if you are using the Classic Agent.

You can suppress the controls for certain classes for .NET, Java SWT, and Windows API-based applications. For example, you might want to ignore container classes to streamline your test cases. Ignoring these unnecessary classes simplifies the object hierarchy and shortens the length of the lines of code in your test scripts and functions. Container classes or ‘frames’ are common in GUI development, but may not be necessary for testing.

The following classes are commonly suppressed during recording and playback:

Technology Domain	Class
.NET	Group
Java SWT	org.eclipse.swt.widgets.Composite org.eclipse.swt.widgets.Group
Windows API-based applications	Group

To suppress specific controls:

1. Click **Options > Class Map**. The **Class Map** dialog box opens.
2. In the **Custom class** field, type the name of the class that you want suppress.
The class name depends on the technology and the extension that you are using. For Windows API-based applications, use the Windows API-based class names. For Java SWT applications, use the fully qualified Java class name. For example, to ignore the **SWT_Group** in a Windows API-based application, type `SWT_Group`, and to ignore to ignore the `Group` class in Java SWT applications, type `org.eclipse.swt.widgets.Group`.
3. In the **Standard class** list, select **Ignore**.
4. Click **Add**. The custom class and the standard class display at the top of the dialog box.

Suppressing Controls (Open Agent)

This functionality is supported only if you are using the Open Agent.

To simplify the object hierarchy and to shorten the length of the lines of code in your test scripts and functions, you can suppress the controls for certain unnecessary classes in the following technologies:

- Win32.
- Java AWT/Swing.
- Java SWT/Eclipse.
- Windows Presentation Foundation (WPF).

For example, you might want to ignore container classes to streamline your test cases.

To suppress specific controls:

1. Click **Options > Recorder**. The **Recording Options** dialog box opens.
2. Click the **Transparent Classes** tab.
3. Type the name of the class that you want to ignore during recording and playback into the text box.

If the text box already contains classes, add the new classes to the end of the list. Separate the classes with a comma. For example, to ignore both the AOL Toolbar and the `_AOL_Toolbar` class, type `AOL Toolbar, _AOL_Toolbar` into the text box.

The `OPT_TRANSPARENT_CLASSES` option is set to true for these classes.

4. Click **OK**. The `OPT_TRANSPARENT_CLASSES` option is set to true for these classes, which means the classes are added to the list of the classes that are ignored during recording and playback.

Configuring Standard Applications

A standard application is an application that does not use a Web browser, such as a Windows application or Java SWT application.

Configure the application that you want to test to set up the environment that Silk Test Classic will create each time you record or replay a test case.

1. Start the application that you want to test.
2. Click **Configure Applications** on the basic workflow bar.

If you do not see **Configure Applications** on the workflow bar, ensure that the default agent is set to the Open Agent.

The **Select Application** dialog box opens.

3. Select the **Windows** tab.
4. Select the application that you want to test from the list.



Note: If the application that you want to test does not appear in the list, uncheck the **Hide processes without caption** check box. This option, checked by default, is used to filter only those applications that have captions.

5. *Optional:* Check the **Create Base State** check box to create a base state for the application under test. By default, the **Create Base State** check box is checked for projects where a base state for the application under test is not defined, and unchecked for projects where a base state is defined. An application's base state is the known, stable state that you expect the application to be in before each test begins execution, and the state the application can be returned to after each test has ended execution. When you configure an application and create a base state, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files location** in the **Runtime Options** dialog box.
6. Click **OK**.
 - If you have checked the **Create Base State** check box, the **Choose name and folder of the new frame file** page opens. Silk Test Classic configures the recovery system and names the corresponding file `frame.inc` by default.
 - If you have not checked the **Create Base State** check box, the dialog box closes and you can skip the remaining steps.
7. Navigate to the location in which you want to save the frame file.
8. In the **File name** text box, type the name for the frame file that contains the default base state and recovery system. Then, click **Save**. Silk Test Classic creates a base state for the application and opens the include file.
9. Record the test case whenever you are ready.



Note: For SAP applications, you must set **Ctrl+Alt** as the shortcut key combination to use. To change the default setting, click **Options > Recorder** and then check the **OPT_ALTERNATE_RECORD_BREAK** check box.

Determining the priorLabel in the Win32 Technology Domain

To determine the priorLabel in the Win32 technology domain, all labels and groups in the same window as the target control are considered. The decision is then made based upon the following criteria:

- Only labels either above or to the left of the control, and groups surrounding the control, are considered as candidates for a priorLabel.
- In the simplest case, the label closest to the control is used as the priorLabel.
- If two labels have the same distance to the control, the priorLabel is determined based upon the following criteria:
 - If one label is to the left and the other above the control, the left one is preferred.
 - If both levels are to the left of the control, the upper one is preferred.
 - If both levels are above the control, the left one is preferred.
- If the closest control is a group control, first all labels within the group are considered according to the rules specified above. If no labels within the group are eligible, then the caption of the group is used as the priorLabel.

Limitations for Testing on Microsoft Windows 10

The following list lists the known limitations for testing on Microsoft Windows 10 (Windows 10):

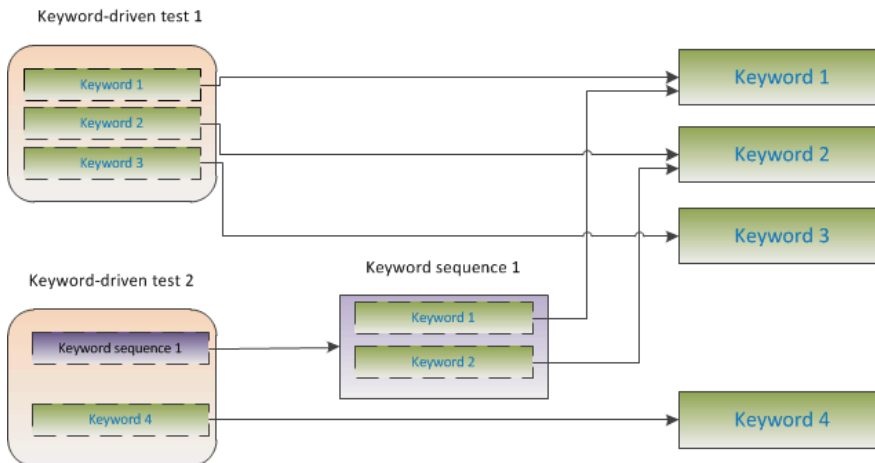
- Silk Test does not support testing Universal Windows Platform (UWP) apps on Windows 10.

Keyword-Driven Tests

This functionality is supported only if you are using the Open Agent.

Keyword-driven testing is a software testing methodology that separates test design from test development and therefore allows the involvement of additional professional groups, for example business analysts, in the test automation process. Silk Central and Silk Test support the keyword-driven testing methodology and allow a very close collaboration between automation engineers and business analysts by having automation engineers develop a maintainable automation framework consisting of shared assets in the form of keywords in Silk Test. These keywords can then be used by business analysts either in Silk Test to create new keyword-driven tests or in Silk Central to convert their existing manual test assets to automated tests or to create new keyword-driven tests.

- A *keyword-driven test* is an executable collection of keywords. A keyword-driven test can be played back just like any other test.
- A *keyword sequence* is a keyword that is a combination of other keywords. Keyword sequences bundle often encountered combinations of keywords into a single keyword, enabling you to reduce maintenance effort and to keep your tests well-arranged.
- A *keyword* is a defined combination of one or more actions on a test object. The implementation of a keyword can be done with various tools and programming languages, for example Java or .NET.



There are two phases required to create keyword-driven tests:

1. Designing the test.
2. Implementing the keywords.

Advantages of Keyword-Driven Testing

This functionality is supported only if you are using the Open Agent.

The advantages of using the keyword-driven testing methodology are the following:

- Keyword-driven testing separates test automation from test case design, which allows for better division of labor and collaboration between test engineers implementing keywords and subject matter experts designing test cases.
- Tests can be developed early, without requiring access to the application under test, and the keywords can be implemented later.
- Tests can be developed without programming knowledge.

- Keyword-driven tests require less maintenance in the long run. You need to maintain the keywords, and all keyword-driven tests using these keywords are automatically updated.
- Test cases are concise.
- Test cases are easier to read and to understand for a non-technical audience.
- Test cases are easy to modify.
- New test cases can reuse existing keywords, which amongst else makes it easier to achieve a greater test coverage.
- The internal complexity of the keyword implementation is not visible to a user that needs to create or execute a keyword-driven test.

Keywords

This functionality is supported only if you are using the Open Agent.

A *keyword* is a defined combination of one or more actions on a test object. The implementation of a keyword can be done with various tools and programming languages, for example Java or .NET. In Silk Test Classic, a keyword is a method with the keyword keyword before the method name. Keywords are saved as keyword assets.

You can define keywords and keyword sequences during the creation of a keyword-driven test and you can then implement them as test methods. You can also mark existing test methods as keywords by adding the keyword keyword before the method name..Keywords are defined as follows:

```
keyword method_name() alias "keyword_name" keyworddescription
"description_text"
```

You can use the alias in the definition to give a name to the keyword that is different than the method name. The alias and the description are optional.

A *keyword sequence* is a keyword that is a combination of other keywords. Keyword sequences bundle often encountered combinations of keywords into a single keyword, enabling you to reduce maintenance effort and to keep your tests well-arranged.

A keyword can have input and output parameters. Any parameter of the test method that implements the keyword is a parameter of the keyword.



Note: You cannot use group keywords in Silk Test Classic.

Example

A test method that is marked as a keyword can look like the following:

```
keyword Login() alias "Login"
... // method implementation
```

or

```
keyword Login(string userName, string password) alias "Login"
keyworddescription "Logs in with the given name and password."
... // method implementation
```

where the keyword logs into the application under test with a given user name and password.



Note: If you are viewing this help topic in PDF format, this code sample might include line-breaks which are not allowed in scripts. To use this code sample in a script, remove these line-breaks.

Creating a Keyword-Driven Test in Silk Test Classic

Before you can create a keyword-driven test in Silk Test Classic, you have to select a project.

This functionality is supported only if you are using the Open Agent.

Use the **Keyword-Driven Test Editor** to combine new keywords and existing keywords into new keyword-driven tests. New keywords need to be implemented as methods in a later step.

1. Click **File > New**. The **New File** dialog box opens.
2. Select **Keyword-driven test** from the file type list.
3. Click **OK**. The **Keyword-Driven Test Editor** opens.
4. Perform one of the following actions:
 - To add a new keyword, type a name for the keyword into the **New Keyword** field.
 - To add an existing keyword, expand the list and select the keyword that you want to add.
5. Press **Enter**.
6. Repeat the previous two steps until the test includes all the keywords that you want to execute.
7. Click **File > Save**. The **Save As** dialog box opens.
8. Type a name for the new test into the **File name** field.
9. Click **Save**.
10. Click **Yes** to add the keyword-driven test to the active project.

Continue with implementing the keywords or with executing the test, if all keywords are already implemented.

Recording a Keyword-Driven Test in Silk Test Classic

This functionality is supported only if you are using the Open Agent.

To record a single keyword, see [Recording a Keyword](#).

To record a new keyword-driven test:

1. In the workflow bar, click **Record Testcase**. The **Start Recording** dialog box opens.
2. Select **Keyword-Driven Test**.
3. Click **Yes** to start recording the keyword-driven test.
4. Select the application state.

During the recording of a keyword-driven test, Silk Test Classic generates a base state keyword with the same name as the application state that you have selected in the **Record Keyword-Driven Test** dialog box. By default, the application state is the `DefaultBaseState`. Silk Test Classic inserts this base state keyword as the first keyword of the keyword-driven test.
5. Click **Start Recording**.
6. If you have not set an application configuration for the current project, select the tab that corresponds to the type of application that you are testing:
 - If you are testing a standard application that does not run in a browser, select the **Windows** tab.
 - If you are testing a web application or a mobile web application, select the **Web** tab.
 - If you are testing a native mobile application, select the **Mobile** tab.
7. To test a standard application, select the application from the list.
8. To test a Web application or a mobile Web application, select one of the installed browsers or mobile browsers from the list.

Specify the Web page to open in the **Enter URL to navigate** text box. If an instance of the selected browser is already running, you can click **Use URL from running browser** to record against the URL currently displayed in the running browser instance.

9. To test a native mobile application (app):

- a) Select the mobile device, on which you want to test the app, from the list.
- b) Click **Browse** to select the app file or enter the full path to the app file into the **Mobile app file** text field.

Silk Test Classic supports HTTP and UNC formats for the path.

Silk Test Classic installs the app on the mobile device or emulator.

10. If you have set an application configuration for the current project and you are testing a Web application, the **Select Browser** dialog box opens. Select the browser on which you want to record the keyword.

11. Depending on the dialog that is open, perform one of the following:

- In the **Select Application** dialog box, click **OK**.
- In the **Select Browser** dialog box, click **Record**.

12. If you have defined a new application configuration, specify a name and location for the new frame file.

13. In the application under test, perform the actions that you want to include in the first keyword.

For information about the actions available during recording, see *Actions Available During Recording*.

14. To specify a name for the keyword, hover the mouse cursor over the keyword name in the **Recording** window and click **Edit**.



Note: Silk Test Classic automatically adds the keyword *Start application* to the start of the keyword-driven test. In this keyword, the applications base state is executed to enable the test to replay correctly. For additional information on the base state, see *Base State*.

15. Type a name for the keyword into the **Keyword name** field.

16. Click **OK**.

17. To record the actions for the next keyword, type a name for the new keyword into the **New keyword name** field and click **Add**. Silk Test Classic records any new actions into the new keyword.

18. Create new keywords and record the actions for the keywords until you have recorded the entire keyword-driven test.

19. Click **Stop**. The **Record Keyword-Driven Test** dialog box opens.

20. Click **Paste to Editor**. The **Update Files** dialog box opens.

21. Click **Paste testcase and update window declaration(s)**. The **Save Keyword Implementation** dialog box opens.

22. Select a file name and a location to save the implementation of the recorded keywords.

23. Click **Save**.

Silk Test Classic creates the new keyword-driven test with all recorded keywords.

Setting the Base State for a Keyword-Driven Test in Silk Test Classic

This functionality is supported only if you are using the Open Agent.

When you execute a keyword-driven test with Silk Test Classic and the keyword-driven test calls a base state keyword, Silk Test Classic starts your AUT from the base state.

During the recording of a keyword-driven test, Silk Test Classic generates a base state keyword with the same name as the application state that you have selected in the **Record Keyword-Driven Test** dialog box. By default, the application state is the `DefaultBaseState`. Silk Test Classic inserts this base state keyword as the first keyword of the keyword-driven test.

Implementing a Keyword in Silk Test Classic

Before implementing a keyword, define the keyword as part of a keyword-driven test.

This functionality is supported only if you are using the Open Agent.

To implement a keyword for reuse in keyword-driven tests:

1. Open a keyword-driven test that includes the keyword that you want to implement.
2. In the **Keyword-Driven Test Editor**, click **Implement Keyword** to the left of the keyword that you want to implement. The **Start Recording** dialog box opens.
3. Perform one of the following actions:
 - To record the keyword, click **Yes**.
 - To create an empty keyword method, click **No**.
4. In the **Create Keyword Script** dialog box, click **Yes** to implement the keyword in a new script file, or click **No** to add the keyword implementation to an existing script file.
5. If you have set an application configuration for the current project and you are testing a Web application, the **Select Browser** dialog box opens. Select the browser on which you want to record the keyword.

You can now record or manually script actions for the keyword.

Recording a Keyword in Silk Test Classic

You can only record actions for a keyword that already exists in a keyword-driven test, not for a keyword that is completely new. To record a new keyword-driven test, see [Recording a Keyword-Driven Test](#).

This functionality is supported only if you are using the Open Agent.

To record the actions for a new keyword:

1. Open a keyword-driven test that includes the keyword that you want to record.
2. In the **Keyword-Driven Test Editor**, click **Implement Keyword** to the left of the keyword that you want to implement. The **Start Recording** dialog box opens.
3. Click **Yes**.
4. Click **Yes**. The **Recording** window opens and Silk Test Classic starts recording the actions for the keyword.
5. In the application under test, perform the actions that you want to test.
For information about the actions available during recording, see *Actions Available During Recording*.
6. Click **Stop**. The **Record Actions on Open Agent** dialog box opens.
7. Click **Paste to Editor**. The **Update Files** dialog box opens.
8. Click **Paste script and update window declaration(s)**.
9. Click **OK**. The **Record Actions on Open Agent** dialog box opens.
10. Click **Close**.

The recorded actions are displayed in the defined 4Test script.

Marking a Test Method in a Script as a Keyword

This functionality is supported only if you are using the Open Agent.

Mark an existing test method in a script as a keyword to reuse the method in keyword-driven tests.

1. Open the script which includes the test method that you want to mark as a keyword.
2. Add `keyword` to the start of the test method.
By default, the keyword name is the name of the test method.
3. *Optional:* You can set a different name for the keyword by adding `alias "KeywordName"` to the start of the test method.

You can now use the test method as a keyword in a keyword-driven test.

Examples

To mark the test method `TestLogin` as a new keyword with the name *Login*, type the following before the start of the test method:

```
keyword TestLogin() alias "Login"
```

To mark the test method `TestLogin` as a new keyword with the name *Login* and with the two input parameters `UserName` and `PassWord`, type the following:

```
keyword TestLogin(string UserName, string PassWord) alias  
"Login" keyworddescription "Logs in with the given name and  
password."  
... // method implementation
```



Note: If you are viewing this help topic in PDF format, this code sample might include line-breaks which are not allowed in scripts. To use this code sample in a script, remove these line-breaks.

Editing a Keyword-Driven Test

This functionality is supported only if you are using the Open Agent.



Note: In Silk Test Classic, you can edit and execute keyword-driven tests that are located in Silk Test Classic, and you can execute keyword-driven tests that are stored in Silk Central. To edit a keyword-driven test, which is stored in Silk Central, open the keyword-driven test in the **Keyword-Driven Test Editor** and click **Edit**.

To edit a keyword-driven test:

1. Open the keyword-driven test in the **Keyword-Driven Test Editor**.
 - a) Click **File > Open** and browse for the keyword-driven test.
If the test is included in a project, you can also expand the **Keyword Tests** node and double-click on the keyword-driven test that you want to edit.
 - b) Click **Keyword-Driven Test**.
2. To add a new keyword to the keyword-driven test:
 - a) Click into the **New Keyword** field.
 - b) Type a name for the new keyword.
 - c) Press **Enter**.

3. To edit an existing keyword, click **Open Keyword** to the left of the keyword.



Note: Silk Central has the ownership of any keyword that has been created in Silk Central, which means any changes that you make to such keywords are saved in Silk Central, not in Silk Test Classic.





4. To remove the keyword from the keyword-driven test, click **Delete Keyword** to the left of the keyword.
The keyword is still available in the **Keywords** window and you can re-add it to the keyword-driven test at any time.

- To save your changes, click **File > Save**.

Managing Keywords in a Test in Silk Central

This functionality is supported only if you are using the Open Agent.

The **Keywords** page enables you to manage the keywords of the selected keyword-driven test. The following actions are possible:

Task	Steps
Opening a test or keyword sequence in Silk Test	Click Open with Silk Test to open the selected test or keyword sequence in Silk Test.
Adding a keyword	<ol style="list-style-type: none"> Click New Keyword at the bottom of the keywords list, or right-click a keyword and select Insert Keyword Above from the context menu. <ul style="list-style-type: none">  Note: You can let Silk Test recommend keywords based on their usage. Toggle the recommendations on or off with Enable Recommendations or Disable Recommendations in the context menu. For additional information, see <i>Which Keywords Does Silk Test Classic Recommend?</i> Select a keyword from the list of available keywords or type a new name to create a new keyword. Click Save. <p>Alternatively, double click an existing keyword in the All Keywords pane on the right or drag and drop it.</p> <ul style="list-style-type: none">  Tip: You can select multiple keywords with Ctrl+Click. When dropping them, they will be sorted in the order that you selected them in.
Deleting a keyword	Click  in the Actions column of the keyword that you want to delete. Click Save .
Changing the order of keywords	Drag and drop a keyword to the desired position. Click Save .
Creating a keyword sequence (a keyword consisting of other keywords)	<ol style="list-style-type: none"> Select the keywords that you want to combine in the keywords list. Use Ctrl+Click or Shift+Click to select multiple keywords. Right-click your selection and click Combine. Specify a Name and Description for the new keyword sequence.
Extracting keywords from a keyword sequence	Right-click a keyword sequence and click Extract keywords . The original keyword sequence is then replaced by the keywords that it contained, but it is not removed from the library. Click Save .
Defining parameters for a keyword sequence	<ol style="list-style-type: none"> Click Parameters above the keywords list. The Parameters dialog box appears. Click Add Parameter. Specify a Name for the new parameter. If the parameter is an outgoing parameter (delivers a value, instead of requiring an input value), check the Output checkbox. Click OK. Click Save.
Editing a draft keyword	<ol style="list-style-type: none"> Click  in the Actions column of the draft keyword that you want to edit.

Task	Steps
	<ol style="list-style-type: none"> 2. Select a Group or specify a new group for the keyword. 3. Type a Description for the keyword. This information is valuable for the engineer who will implement the keyword. 4. Click OK. 5. <i>Optional:</i> Click into a parameter field to add parameters for the keyword. If the keyword is implemented with Silk Test, these parameters will appear in the generated code stub. 6. Click Save.
Searching for a keyword	<p>Use the search field in the Keywords window to find a specific keyword. When you enter alphanumeric characters, the list is dynamically updated with all existing matches. Tips for searching:</p> <ul style="list-style-type: none"> • The search is case-insensitive: doAction will find doaction and DOAction. • Enter only capital letters to perform a so-called <i>CamelCase</i> search: ECD will find Enter Car Details, Enter Contact Details and EnterContactDetails. • Keyword and group names are considered: test will find all keywords that contain test and all keywords in groups where the group name contains test. • ? replaces 0-1 characters: user?test will find userTest and usersTest. • * replaces 0-n characters: my*keyword will find myKeyword, myNewKeyword and my_other_keyword. • .<string> only searches in keyword names: .keyword will find all keywords that contain keyword. • Use quotes to search for an exact match: 'Keyword' will find Keyword and MyKeyword, but not keyword.

Which Keywords Does Silk Test Classic Recommend?

When you add keywords to a keyword-driven test or a keyword sequence in the **Keyword-Driven Test Editor**, Silk Test Classic recommends existing keywords which you might want to use as the next keyword in your test. The recommended keywords are listed on top of the keywords list, and are indicated by a bar graph, with the filled-out portion of the graph corresponding to how much Silk Test Classic recommends the keyword.

Silk Test Classic recommends the keywords based on the following:

- When you add the first keyword to a keyword-driven test or a keyword sequence, Silk Test Classic searches for similar keywords that are used as the first keyword in other keyword-driven tests or keyword sequences. The keywords that are used most frequently are recommended higher.
- When you add additional keywords to a keyword-driven test or a keyword sequence, which already includes other keywords, Silk Test Classic recommends keywords as follows:
 - If there are keywords before the position in the keyword-driven test or the keyword sequence, to which you add a new keyword, Silk Test Classic compares the preceding keywords with keyword combinations in all other keyword-driven tests and keyword sequences and recommends the keywords that most frequently follow the preceding combination of keywords.
 - If there are no keywords before the position in the keyword-driven test or the keyword sequence, but there are keywords after the current position, then Silk Test Classic compares the succeeding

keywords with keyword combinations in all other keyword-driven tests and keyword sequences and recommends the keywords that most frequently precede the succeeding combination of keywords.

- Additionally, Silk Test Classic takes into account how similar the found keywords are. For example, if both the name and group of two keywords match, then Silk Test Classic recommends these keywords higher in comparison to two keywords for which only the name matches.
- If you have established a connection with Silk Central, any keywords included in keyword-driven tests, which belong to the keyword library that corresponds to the current project, are also considered.

Using Parameters with Keywords

A keyword or a keyword sequence can have input and output parameters. This topic describes how you can handle these parameters with Silk Test Classic.

In the **Keyword-Driven Test Editor**, you can view any defined parameters for a keyword or a keyword sequence and you can edit the parameter values.

In the **Keywords** window, you can see which parameters are assigned to a keyword or a keyword sequence when you hover the mouse cursor over the keyword or keyword sequence.

Input parameters for simple keywords

You can define and use input parameters for keywords in the same way as for any other test method.

The following code sample shows how you can define the keyword `SetUserDetails` with the two input parameters `userName` and `password`:

```
keyword SetUserDetails(String userName, String password)
    ...
```

Output parameters for simple keywords

You can define a return value or one or more output parameters for a keyword. You can also use a combination of a return value and one or more output parameters.

The following code sample shows how you can define the keyword `GetText` that returns a string:

```
keyword GetText()
    return "text"
```

The following code sample shows how you can define the keyword `GetUserDetails` with the two output parameters `userName` and `password`:

```
keyword GetUserDetails(out String userName null, out String password)
    userName="name"
    password="password"
```

Parameters for keyword sequences

You can define or edit the parameters for a keyword sequence in the **Parameters** dialog box, which you can open if you click **Parameters** in the **Keyword Sequence Editor**.

Example: Keywords with Parameters

This topic provides an example of how you can use keywords with parameters.

As a first step, create a keyword-driven test which contains the keywords that you want to use. You can do this by recording an entire keyword-driven test, or by creating a new keyword-driven test and by adding the keywords in the keyword-driven test editor.

In this example, the keyword-driven test includes the following keywords:

- Start application** This is the standard keyword that starts the AUT and sets the base state.
- Login** This keyword logs into the AUT with a specific user, identified by a user name and a password.
- GetCurrentUser** This keyword returns the name of the user that is currently logged in to the AUT.
- AssertEquals** This keyword compares two values.
- Logout** This keyword logs the user out from the AUT.

The next step is to add the parameters to the keywords. To do this, open the test scripts of the keywords and add the parameters to the methods.

To add the input parameters `UserName` and `Password` to the keyword `Login`, change

```
keyword Login()
...
```

to

```
keyword Login(STRING UserName, STRING Password)
...
```

To add the output parameter `UserName` to the keyword `GetCurrentUser`, change

```
keyword GetCurrentUser()
...
```










to

```
keyword GetCurrentUser(out STRING CurrentUser)
...
```

The keyword-driven test in the **Keyword-Driven Test Editor** should look similar to the following:

		Keyword	Parameters	
1	 	Start application		
2	 	Login	<i>UserName</i> <i>Password</i>	
3	 	GetCurrentUser	<i>CurrentUser ←</i>	
4	 	AssertEquals	<i>Expected</i> <i>Actual</i>	
5	 	Logout		

Now you can specify actual values for the input parameters in the **Keyword-Driven Test Editor**. To retrieve the value of the output parameter `UserName` of the keyword `GetCurrentUser`, provide a variable, for example `${current user}`. You can then pass the value that is stored in the variable to subsequent keywords.

		Keyword	Parameters	
1	 	Start application		
2	 	Login	<i>UserName</i> <i>Password</i>	
3	 	GetCurrentUser	<code>\${current user}</code>	
4	 	AssertEquals	John Smith <code>\${current user}</code>	
5	 	Logout		

Combining Keywords into Keyword Sequences

This functionality is supported only if you are using the Open Agent.

Use the **Keyword-Driven Test Editor** to combine keywords, which you want to execute sequentially in multiple keyword-driven tests, into a keyword sequence.

1. Open the keyword-driven test that includes the keywords that you want to combine.
2. In the **Keyword-Driven Test Editor**, press and hold down the `Ctrl` key and then click the keywords that you want to combine.
3. Right-click on the selection and click **Combine**. The **Combine Keywords** dialog box opens.
4. Type a name for the new keyword sequence into the **Name** field.
5. *Optional:* Type a description for the new keyword sequence into the **Description** field.
6. Click **Combine**.

The new keyword sequence opens and is also displayed in the **Keywords** window. You can use the keyword sequence in keyword-driven tests.



Note: Like any other keyword, you cannot execute a keyword sequence on its own, but only as part of a keyword-driven test.

Replaying a Keyword-Driven Test with Specific Variables

Before you can set the values of variables for the execution of a keyword-driven test, you have to create the project.

When executing keyword-driven tests that are part of an automation framework and that are managed in a test management tool, for example Silk Central, you can set the values of any variables that are used for the execution of the keyword-driven test in Silk Test Classic. To set the values of global variables for the entire project, which means that these values are used whenever a Silk Test Classic user executes the keyword-driven test assets in this project, perform the following actions:

1. Open the project that includes the keyword-driven test.
2. Right-click the folder **Data** of the project and select **New File**. A new untitled file is opened in the editor.
3. Add new lines to the file to specify the variables.

The format for a new variable is:

```
name=value
```

For example, to specify the two variables *user* and *password*, type the following:

```
user=John  
password=john5673
```

4. In the menu, click **File > Save**.
5. Type `globalvariables.properties` into the **File name** field.
6. Click **Save**.

Whenever the keyword-driven test is executed from Silk Test Classic, the variables are used.



Note: When you replay a keyword-driven test, which is located in Silk Central, from Silk Test Classic, the result file (.xlg) is saved into the folder of the current Silk Test Classic project.

Integrating Silk Test Classic with Silk Central

Integrate Silk Test Classic and Silk Central to enable collaboration between technical and less-technical users.

This functionality is supported only if you are using the Open Agent.

When Silk Test Classic and Silk Central are integrated and a library with the same name as the active Silk Test Classic project exists in Silk Central, the **Keywords** view under **Window > Keywords** displays all keywords from the Silk Central library in addition to any keywords defined in the active Silk Test Classic project.



Note: The Silk Central connection information is separately stored for every Silk Test Classic user, which means every Silk Test Classic user that wants to work with keywords and keyword sequences from Silk Central must integrate Silk Test Classic with Silk Central.

Integrating Silk Test Classic with Silk Central provides you with the following advantages:

- Test management and execution is handled by Silk Central.
- Keywords are stored in the Silk Central database (upload library) and are available to all projects in Silk Central.
- Manual tests can be directly automated in Silk Central and the created keyword-driven tests can be executed in Silk Test Classic from Silk Central.



Note: In Silk Test Classic, you can edit and execute keyword-driven tests that are located in Silk Test Classic, and you can execute keyword-driven tests that are stored in Silk Central. To edit a keyword-driven test, which is stored in Silk Central, open the keyword-driven test in the **Keyword-Driven Test Editor** and click **Edit**.

1. From the menu, select **Options > Silk Central Configuration**. The **Silk Central Connection** dialog box opens.
2. Type the URL of your Silk Central server into the **URL** field.
For example, if the Silk Central server name is *sctm-server*, and the port for Silk Central is 13450, type `http://sctm-server:13450`.
3. Type a valid user name and password into the corresponding fields.
4. Click **Verify** to verify if Silk Test Classic can access the Silk Central server with the specified user.
5. Click **OK**.

Implementing Silk Central Keywords in Silk Test Classic

Before implementing Silk Central keywords, define the keywords as part of a keyword-driven test in Silk Central.

This functionality is supported only if you are using the Open Agent.

To implement a Silk Central keyword in Silk Test Classic:

1. Create a project in Silk Test Classic with the same name as the keyword library in Silk Central, which includes the keyword-driven test.
2. If the keyword library in Silk Central has no type assigned, click **Upload Keyword Library** in the toolbar to set the library type.
3. *Optional:* To implement a specific keyword in Silk Test Classic from Silk Central, open the **Keywords** tab of the library in Silk Central and click **Implement with Silk Test** in the **Actions** column of the keyword.
4. In Silk Test Classic, open the project and select the **Keywords** tab.

5. In the **Keywords** tab, double-click the keyword-driven test.
To update the **Keywords** tab with any changes from Silk Central, click **Refresh**.
6. In the toolbar, click **Record Testcase**.
7. If you have set an application configuration for the current project and you are testing a Web application, the **Select Browser** dialog box opens. Select the browser on which you want to record the keyword.
8. Record the actions for the first unimplemented keyword.
9. When you have recorded all the actions for the current keyword, click **Next Keyword**.
10. To switch between keywords in the **Recording** window, click **Previous Keyword** and **Next Keyword**.
11. Click **Stop**. The **Record Keyword-Driven Test** dialog box opens.



Note: You cannot delete keywords or change the sequence of the keywords in a keyword-driven test from Silk Central, as these tests are read only in Silk Test Classic.

Uploading a Keyword Library to Silk Central

This functionality is supported only if you are using the Open Agent.

To work with Silk Central, ensure that you have configured a valid Silk Central location. For additional information, see [Integrating Silk Test Classic with Silk Central](#).

To automate manual tests in Silk Central, upload keywords that you have implemented in a Silk Test Classic project as a keyword library to Silk Central, where you can then use the keywords to automate manual tests.



Note: Ensure that the project does not reference files that are located outside of the project directory.

1. In Silk Test Classic, select the project in which the keyword-driven tests reside.
2. Ensure that a library with the same name exists in Silk Central (**Tests > Libraries**).
3. *Optional:* Provide a description of the changes to the keyword library.
4. *Optional:* Click **Configure** to configure the connection to Silk Central.
5. *Optional:* To see which libraries are available in the connected Silk Central instance, click on the link.
6. Click **Upload**.



Caution: If the keyword library in Silk Central is already assigned to a different automation tool or another Silk Test client, you are asked if you really want to change the type of the keyword library. Upload the library only if you are sure that you want to change the type.

Silk Test Classic creates a keyword library out of all the keywords that are implemented in the project. Then Silk Test Classic saves the keyword library with the name `library.zip` into the output folder of the project. The library is validated for consistency, and any changes which might break existing tests in Silk Central are listed in the **Upload Keyword Library to Silk Central** dialog box. Finally, Silk Test Classic uploads the library to Silk Central. You can now use the keywords in Silk Central. Any keyword-driven tests in Silk Central, which use the keywords that are included in the keyword library, automatically use the current implementation of the keywords.

Uploading a Keyword Library to Silk Central from the Command Line

Upload an external keyword library to Silk Central from a Java-based command line to integrate Silk Central and your keyword-driven tests into your continuous integration build system, for example Jenkins.

To upload your keyword library to Silk Central from a Java-based command line:

1. Select **Help > Tools** in Silk Central and download the **Java Keyword Library Tool**.
2. Call the command line tool that is contained in the downloaded `jar` file with the following arguments:
 - `java`
 - `-jar com.borland.silk.keyworddriven.jar`
 - `-upload`
 - `Library` name of the library in Silk Central to be updated, or created if it does not yet exist.
 - `Package` name of the library package (zip archive) to be uploaded.
 - `Hostname:port` of the Silk Central front-end server.
 - `Username` of the Silk Central user.
 - `Password` of the Silk Central user.
 - `Update` information, describing the changes that were applied to the library, in quotes.
 - `[-allowUsedKeywordDeletion]`, an optional flag to allow the deletion of keywords that are used in a test or keyword sequence. By default, an error is raised if used keywords are attempted to be deleted.

The following example outlines the command line to upload a library to Silk Central:

```
java -jar com.borland.silk.keyworddriven.jar -upload
"My library" "./output/library.zip" silkcentral:19120 scLogin
scPassword "Build xy: Implemented missing keywords"
```

Searching for a Keyword

This functionality is supported only if you are using the Open Agent.

Use the search field in the **Keywords** window to find a specific keyword. When you enter alphanumeric characters, the list is dynamically updated with all existing matches. Tips for searching:

- The search is case-insensitive: `doAction` will find `doaction` and `DOAction`.
- Enter only capital letters to perform a so-called *Camel/Case* search: `ECD` will find `Enter Car Details`, `Enter Contact Details` and `EnterContactDetails`.
- Keyword and group names are considered: `test` will find all keywords that contain `test` and all keywords in groups where the group name contains `test`.
- `?` replaces 0-1 characters: `user?test` will find `userTest` and `usersTest`.
- `*` replaces 0-n characters: `my*keyword` will find `myKeyword`, `myNewKeyword` and `my_other_keyword`.
- `.<string>` only searches in keyword names: `.keyword` will find all keywords that contain `keyword`.
- Use quotes to search for an exact match: `'Keyword'` will find `Keyword` and `MyKeyword`, but not `keyword`.

Filtering Keywords

To find a specific keyword in the current project, you can filter the keywords that are displayed in the **Keywords** window. If an integration with Silk Central is configured, the result includes the relevant keywords from Silk Central.

1. In the menu, click **Keywords**. The **Keywords** window opens.
2. In the **Keywords** window, type the name of the keyword that you are searching for into the search field. The **Keywords** window lists all keywords in the current project with the given name.
3. *Optional:* To edit a keyword, hover the mouse cursor over the keyword in the **Keywords** window and click **Go to implementation**.

Using Advanced Techniques with the Open Agent

This section describes advanced techniques for testing your applications with Silk Test Classic and the Open Agent.

Starting from the Command Line

This section describes how you can start Silk Test Classic from the command line.

Starting Silk Test Classic from the Command Line

You can start the Silk Test Classic executable program from the command line by:

- Clicking **Run** in the **Start** menu.
- Using the command-line prompt in a DOS window or batch file.

The syntax is:

```
partner [-complog filename] [-m mach] [-opt optionset.opt] [-p mess] [-proj filename [-base filename]] [[-q] [-query query name] [-quiet] [-r filename] [-resexport] [-resextract] [-r] scr.t/suite.s/plan.pln/link.lnk [args]]
```

The *filename* specified for various options expects the file to be located in the working directory. The default location is the Silk Test Classic installation directory, C:\Program Files (x86)\Silk\SilkTest. If you want to use a file that is located in another directory, you must specify the full path in addition to the filename.

Options

The `partner` command can be called with the following options:

args

Optional arguments to a script file. You can access the arguments using the `GetArgs` function and use them with your scripts.

If you pass arguments in the command line, the arguments provided in the command line are used and any arguments specified in the currently loaded options set are not used. To use the arguments in the currently loaded options set, do not specify arguments in the command line.

For additional information, see *Passing arguments to a script*.

-complog

Tells Silk Test Classic to log compilation errors to a file you specify. Enter the argument as `-complog filename`. For example:

```
partner [-complog c:\testing\logtest1.txt]
```

If you include this argument, Silk Test Classic checks that the specified file exists each time that you compile. If the file does not exist, Silk Test Classic creates and opens the file. If the file exists, Silk Test Classic opens it and adds the information. The number of errors is written in the format `n error(s)`, for example `0 errors`, `1 error`, or `50 errors`. Compilation errors are written to the error log file as they are displayed in the **Errors**

window. The error log file is automatically saved and closed when Silk Test Classic finishes writing errors to it.

-m Specifies the target machine. The default is the current machine. Call the 4Test built-in function `Connect` to connect to a different machine at runtime.

In order to use the `-m` switch, you need to have the **Network setting** of the **Runtime Options** dialog box set to `TCP/IP` or `NetBIOS`. If this is set to `'(disabled)'`, the target machine is ignored. To set the **Network setting**, either set it interactively in the **Runtime Options** dialog box before running from the command line, or save the setting in an option set and add the `'-opt <option set>'` argument to the command line.

-opt Specifies an options set. Must be followed by the path of the `.opt` file that you want to use.

-p Provided for use with a Windows shell program that is running Silk Test Classic as a batch task. This option enables another Windows program to receive a message containing the number of errors that resulted from the run. Silk Test Classic broadcasts this message using the Windows `PostMessage` function, with the following arguments:

- `hWnd = HWND_BROADCAST`
- `uiMsg = RegisterWindowMessage (mess)`
- `wParam = 0`
- `lParam = number of errors`

To take advantage of the `-p` option, the shell program that runs Silk Test Classic should first register the message (`mess`), and should look for the message while Silk Test Classic is running.

-proj Optional argument specifying the project file or archived project to load when starting Silk Test Classic or Silk Test Classic Runtime. For example:

```
partner -proj d:\temp\testapp.vtp -r agent.pln
```

`-base` is an optional argument to `-proj`. You use the `base` argument to specify the location where you want to unpack the package contents. For example, `partner -proj d:\temp\testapp.stp -base c:\rel30\testapp` unpacks the contents of the package to the `c:\rel30\testapp` directory.

-q Quits Silk Test Classic after the script, suite, or test plan completes.

-query Specifies a query. Must be followed by the name of a saved query. Tells Silk Test Classic to perform an **Include > Open All**, then **Testplan > Mark By Named Query**, then **Run > Marked Tests**.

-quiet Starts Silk Test Classic in quiet mode, which prevents any pop-up dialog boxes from displaying when Silk Test Classic starts up.

The quiet option is particularly useful if you are doing unattended testing where a user is not available to respond to any pop-up dialog boxes that may display.

-r Must be the last option specified, followed only by the name of a Silk Test Classic file to open. This includes files such as script (and, optionally, arguments that the script takes), a suite, test plan, or link file. If you specify a link file, tells Silk Test Classic to resolve the link and attempt to open the link target. Otherwise, tells Silk Test Classic to run the specified script, suite, or test plan, optionally passing args as arguments to a script file. For example, `partner -proj d:\temp\testapp.stp -base c:\rel30\testapp -r Agent.pln` unpacks the archive from the `temp` subdirectory into the `c:\rel30\testapp` subdirectory and then loads and executes the `Agent.pln` file.

- resexport** Tells Silk Test Classic to export a one line summary of the most recent results sets to `.rex` files automatically. Specifying `-resexport` has the same effect as if each script run invokes the `ResExportOnClose` function during its execution.
- resextract** Tells Silk Test Classic to extract all information from the most recent results sets to a `.txt` file. Both the Silk Test Classic Extract menu command and the `-resextract` option create UTF-8 files.
- script.t/
suite.s/
plan.pln/
link.lnk** The name of the Silk Test Classic script, suite, test plan, or link file to load, run, or open.

Examples

- To load Silk Test Classic, type the following:
`partner`
- To run the `test.s` suite, type the following:
`partner -r test.s on system "sys1"`
- To run the `test.t` script, type the following:
`partner -m sys1 -r test.t`
- To run the `test.t` script with arguments, type the following:
`partner -r test.t arg1 arg2`
- To run the tests marked by the query named `query3` in `tests.pln`, type the following:
`partner -query query3 -r tests.pln`
- To run `tests.pln` and to export the most recent results set from `tests.res` to `tests.rex`, type the following:
`partner -q -resexport -r tests.pln`
- To edit the `test.inc` include file, type the following:
`partner test.inc`


Recording a Test Frame

This section describes how you can record a test frame.

Overview of Object Files

Object files are the compiled versions of include (`.inc`) or script (`.t`) files. Object files are saved with an "o" at the end of the extension, for example, `.ino`, or `.to`. Object files cannot be edited; the only way to change compiled objects is to recompile the include or script file. When you save a script or include file, a source file and an object file are saved. Object files are not platform-specific; you can use them on all platforms that Silk Test Classic supports.

In order for Silk Test Classic to run a script or include file that is in source form, it must compile it, which can be time-consuming. Object files, on the other hand, are ready to run.

 **Note:** You cannot call objects that exist in the object file (`.to`) from a test plan; you must have the script file (`.t`).

To disable saving object files during compilation, the **AutoComplete** options on the **General Options** dialog box as well as the **Save object files during compilation** option on the **Runtime Options** dialog box need to be unchecked.

Silk Test Classic always uses object files if they are available. When you open a script file or an include file, Silk Test Classic loads the corresponding object file as well, if there is one. If the object file is not older than the source file, Silk Test Classic does not recompile the source file. The script is ready to run. If the source file is more recent, Silk Test Classic recompiles the source file before the script is run. If you then later save the source file, Silk Test Classic automatically saves a new object file.

If a file is loaded during compilation, that is, if you include a file in another file that is being compiled, Silk Test Classic loads only the object file, if it exists and is newer than the corresponding source file.

Object files may not be backward-compatible, although sometimes they will be. Specifically, object files will not work with versions of Silk Test Classic for which the list of GUI/browser types is different than for the version used to compile the object file. The list is in `4Test.inc`. For example, object files created before 'mswpx' was added as the GUI type for Windows XP cannot be used with ST5.5 SP3, which includes the 'mswpx' GUI type.

If you are using a `.ino` file, but during compilation Silk Test Classic displays a message that the corresponding `.inc` file is missing, then you may be experiencing the object file version incompatibility explained in the preceding paragraph.

Advantages of Object Files

Advantages of object files include:

- Because object files are ready to run, they do not need to be recompiled if the source file has not changed. This can save you a lot of time. If your object file is more recent than your source file, the source file does not need to be recompiled each time the file is first opened in a session; the object file is used as is.
- You can distribute object files without having to distribute the source equivalents. So if you have built some tests and include files that you want to distribute but don't want others to see the sources, you can distribute only the object files.

Since an object file cannot be run directly:

- Define the code you want to "hide" in an include file, which will be compiled into an `.ino` object file.
- Call those functions from an ordinary script file.
- Distribute the `.t` script file and the compiled `.ino` include file. Users can open and run the script file as usual, through **File > Run**.

Here's a simple example of how you might distribute object files so that others cannot see the code.

In file `test.inc`, place the definition of a function called `TestFunction`. When you save the file, the entire include file is compiled into `test.ino`.

```
TestFunction ()
    ListPrint (Desktop.GetActive ())
```

In the file `test.t` use the `test.inc` include file. Silk Test Classic will load the `.ino` equivalent. Call `TestFunction`, which was defined in the include file.

```
use "test.inc"

main ()
    TestFunction () // call the function
```

Distribute `test.t` and `test.ino`. Users can open `test.t` and run it but do not have access to the actual routine, which resides only in compiled form in `test.ino`.

Object File Locations

By default, an object file is read from and written to the same directory as its corresponding source file. But you can specify different directories for object files.

Specifying `d:\obj` in the **Objfile Path** text box of the **Runtime Options** dialog box tells Silk Test Classic to read and write all object files in the `d:\obj` directory, regardless of where the source files are located.

Specifying `obj` in the **Objfile Path** text box tells Silk Test Classic to read and write an object file in the directory `obj` that is a subdirectory of the directory containing the source file. In this scenario, each directory of source files will have a different directory of object files. For example, if a source file is in `d:\src`, its corresponding object file would be read from and written to `d:\src\obj`.

You can specify several directories in the **Objfile Path** text box. New files are written to the first directory specified. Silk Test Classic searches the directories in the order in which you have specified them to find existing files and will subsequently re-save existing files in the same directory where it found them.

Specifying where Object Files Should be Written To and Read From

By default, an object file is read from and written to the same directory as its corresponding source file. But you can specify different directories for object files. To specify where object files are written to and read from:

1. Click **Options > Runtime**.
2. Specify a directory in the **Objfile Path** text box.
 - Leave the text box empty if you want to store object files in the same directory as their corresponding source files.
 - Specify an absolute path if you want to store all object files in the same directory.
 - Specify a relative path if you want object files to be stored in a directory relative to the directory containing the source files.
3. Click **OK**.

Object files are saved in the location you specify here. In addition, Silk Test Classic will try to find object files in these locations. If it fails to find an object file, it will look in the directory containing the source file.

Declarations

This section describes declarations.

GUI Specifiers

Where Silk Test Classic can detect a difference from one platform to the next, it automatically inserts a **GUI-specifier** in a window declaration to indicate the platform, for example `msw`.

For a complete list of the valid GUI specifiers, see *GUI TYPE data type*.

Overview of Dialog Box Declarations

The declarations for the controls contained by a dialog box are nested within the declaration of the dialog box to show the GUI hierarchy.

The declarations for menus are nested (indented) within the declaration for the main window, and the declarations for the menu items are nested within their respective menus. This nesting denotes the hierarchical structure of the GUI, that is, the parent-child relationships between GUI objects. Although a dialog box is not physically contained by the main window, as is true for menus, the dialog box nevertheless logically belongs to the main window. Therefore, a parent statement within each dialog box declaration is used to indicate that it belongs to the main window of the application.

In the sample Text Editor application, `MainWin` is the parent of the `File` menu. The `File` menu is considered a child of the `MainWin`. Similarly, all the menu items are child objects of their parent, the `File` menu. A child object belongs to its parent object, which means that it is either logically associated with the parent or physically contained by the parent.

Because child objects are nested within the declaration of their parent object, the declarations for the child objects do not need to begin with the reserved word `window`.

Classic Agent Example

The following example from the Text Editor application shows the declarations for the **Find** dialog box and its contained controls:

```
window DialogBox Find
  tag "Find"
  parent TextEditor
  StaticText FindWhatText
    multitag "Find What:"
    "$65535"
  TextField FindWhat
    multitag "Find What:"
    "$1152"
  CheckBox CaseSensitive
    multitag "Case sensitive"
    "$1041"
  StaticText DirectionText
    multitag "Direction"
    "$1072"
  RadioList Direction
    multitag "Direction"
    "$1056"
  PushButton FindNext
    multitag "Find Next"
    "$1"
  PushButton Cancel
    multitag "Cancel"
    "$2"
```

Open Agent Example

The following example from the Text Editor application shows the declarations for the **Find** dialog box and its contained controls:

```
window DialogBox Find
  locator "Find"
  parent TextEditor
  TextField FindWhat
    locator "@caption='Find What:' or @windowId='65535'"
  StaticText FindWhatText
    locator "@caption='Find What:' or @windowId='1152'"
  CheckBox CaseSensitive
    locator "@caption='Case sensitive' or @windowId='1041'"
  StaticText DirectionText
    locator "@caption='Direction' or @windowId='1072'"
  RadioList Direction
    locator "@caption='Direction' or @windowId='1056'"
  PushButton FindNext
    locator "@caption='Find Next' or @windowId='1'"
  PushButton Cancel
    locator "@caption='Cancel' or @windowId='2'"
```

Main Window and Menu Declarations

The main window declaration

The main window declaration begins with the 4Test reserved word `window`. The term `window` is historical, borrowed from operating systems and window manager software, where every GUI object, for example main windows, dialogs, menu items, and controls, is implemented as a window.

As is true for all window declarations, the declaration for the main window is composed of a class, identifier, and tag or locator.

Classic Agent Example

The following example shows the beginning of the default declaration for the main window of the Text Editor application:

```
window MainWin TextEditor
  multitag "Text Editor"
    "$C:\PROGRAMFILES\\SILKTEST
\TEXTEDIT.EXE"
```

Part of Declaration	Value for TextEditor's main window.
Classes	MainWin
Identifier	TextEditor
Tag	Two components in the multiple tag: <ul style="list-style-type: none">" Text Editor "—The application's caption" executable path "—The full path of the executable file that invoked the application

Open Agent Example

The following example shows the beginning of the default declaration for the main window of the Text Editor application:

```
window MainWin TextEditor
  locator "Text Editor"
```

Part of Declaration	Value for TextEditor's main window.
Classes	MainWin
Identifier	TextEditor

Part of Declaration	Value for TextEditor's main window.
Locator	" Text Editor "—The application's caption

sCmdLine and wMainWindow constants

When you record the declaration for your application's main window and menus, the *sCmdLine* and *wMainWindow* constants are created. These constants allow your application to be started automatically when you run your test cases.

The *sCmdLine* constant specifies the path to your application's executable. The following example shows an *sCmdLine* constant for a Windows environment:

```
mswnt const sCmdLine = "c:\program files\<<SilkTest install directory>\silktest\textedit.exe"
```

The *wMainWindow* constant specifies the 4Test identifier for the main window of your application. For example, here is the definition for the *wMainWindow* constant of the Text Editor application on all platforms:

```
const wMainWindow = TextEditor
```

Menu declarations

When you are working with the Classic Agent, the following example from the Text Editor application shows the default main window declaration and a portion of the declarations for the File menu:

```
window MainWin TextEditor
  multitag "Text Editor"
    "$C:\PROGRAM FILES\<<SilkTest install directory>\SILKTEST\TEXTEDIT.EXE"
    .
    .
    Menu File
      tag "File"
      MenuItem New
        multitag "New"
          "$100"
```

Menus do not have window IDs, but menu items do, so by default menus are declared with the `tag` statement while menu items are declared with the `multitag` statement.

When you are working with the Open Agent, the following example from the Text Editor application shows the default main window declaration and a portion of the declarations for the File menu:

```
window MainWin TextEditor
  locator "Text Editor"
  .
  .
  Menu File
    locator "File"
    MenuItem New
      locator "@caption='New' or windowId='100'"
```

Window Declarations

A window declaration specifies a cross-platform, logical name for a GUI object, called the *identifier*, and maps the identifier to the actual name of the object, which is called the *tag* or *locator*. Because

your test cases use logical names, if the actual name of the object changes on the current GUI, on another GUI, or in a localized version of the application, you only need to change the tag in the window declarations. You do not need to change any of your scripts.

You can add variables, functions, methods, and properties to the basic window declarations recorded by Silk Test Classic. For example, you can add variables to a dialog box declaration that specify what the tab sequence is, what the initial values are, and so on. You access the values of variables at runtime as you would a field in a record.

After you record window declarations for the GUI objects in your application and insert them into a declarations file, called an include file (* .inc), Silk Test Classic references the declarations in the include file to identify the objects named in your test scripts. You tell Silk Test Classic which include files to reference through the **Use Files** field in the **Runtime Options** dialog box.

Improving Silk Test Classic Window Declarations

The current methodology for identifying window declarations in Microsoft Windows-based applications during a recording session is usually successful. However, some applications may require an alternate approach of obtaining their declarations because their window objects are invisible to the Silk Test Recorder. You can try any of the following:

- Turning on Accessibility - use this if during a session started with the Recorder, Silk Test Classic is unable to recognize objects within a Microsoft Windows-based application. This functionality is available only for projects or scripts that use the Classic Agent.
- Defining a new window - use this if turning on Accessibility does not help Silk Test Classic to recognize the objects. This functionality is available only for projects or scripts that use the Classic Agent.
- Creating a test case that uses dynamic object recognition - use this to create test cases that use XPath queries to find and identify objects. Dynamic object recognition uses a **Find** or **FindAll** method to identify an object in a test case. This functionality is available only for projects or scripts that use the Open Agent.

Improving Object Recognition by Defining a New Window

If Silk Test Classic is having difficulty recognizing objects in Internet Explorer or Microsoft Office applications, try enabling Accessibility. If that does not help improve recognition, try defining a new window.

How defined windows works

When you use Defined Window, you use the mouse pointer to draw a rectangle around the object that Silk Test Classic cannot record and then assign a name to the object. When you save your work, Silk Test Classic stores the name and the object's coordinates in a test script. When you replay the script, Silk Test Classic uses a `click()` method on the center of the area you have specified.

Notes

- Defining a new window is only available for projects or scripts that use the Classic Agent.
- Defining a new window is not available for Java applications or applets.
- Defined Window does not support nesting of defined objects.
- Defined Window is location-based and uses pixel coordinates to locate the object in the parent window. Thus, if the layout of your parent window changes and/or the object's coordinates change frequently, you may need to re-define the window in order for Silk Test Classic to correctly declare the object.
- If you draw a rectangle around an unrecognized object, but also include an object that Silk Test Classic easily recognizes, Silk Test Classic records both and lists the easily recognized object first.

Recording Window Declarations for the Main Window and Menu Hierarchy

1. Start your application.
2. Click **File > New** in Silk Test Classic.
3. Click **Test Frame** and then click **OK**. Silk Test Classic displays the **New Test Frame** dialog box.
4. If you are using the Open Agent, follow the appropriate wizard to select your application, depending on whether you want to test an application that uses a Web browser or not. When you have stepped through the wizard, the **Choose name and folder of the new frame file** dialog box opens.
5. In the **Frame filename** (Classic Agent) or the **File name** (Open Agent) text box, accept the default test frame name (`frame.inc`), or type a new name.

By default, Silk Test Classic names the new test frame file `frame.inc`, denoting it is an include file that contains declarations. If you change the default name of the file, make sure to include the file extension `.inc` in the new file name. If you do not, the file is not identified to Silk Test Classic as an include file and Silk Test Classic will give it a `.txt` extension and report a compilation error when you click **OK** to create the file.

6. If you are using the Classic Agent, select your application from the **Application** list box.
The **Application** list box displays all applications that are open and not minimized. If your test application is not listed, click **Cancel**, open your application, and click **File > New** again.
7. Click **OK** (Classic Agent) or **Save** (Open Agent). Silk Test Classic creates the new test frame file. Window declarations display in the test plan editor, which means that the declarations for individual GUI objects can be expanded to show detail, collapsed to hide detail, and edited if necessary.

Use the member-of Operator to Access Data

Use the member-of operator (`.`) to reference the data defined in a window declaration. For example, if a script needs to know which control should have focus when the **Find** dialog box is first displayed, it can access this data from the window declaration with this expression:

```
Find.lwTabOrder[1]
```

Similarly, to set focus to the third control in the list:

```
Find.lwTabOrder[3].SetFocus ( )
```

Overview of Identifiers

When you record test cases, Silk Test Classic uses the window declarations in the test frame file to construct a unique identifier, called a fully qualified identifier, for each GUI object. The fully-qualified identifier consists of the identifier of the object, combined with the identifiers of the object's ancestors. In this way, the 4Test commands that are recorded can manipulate the correct object when you run your test cases.

If all identifiers were unique, this would not be necessary. However, because it is possible to have many GUI objects with the same identifier, for example the **OK** button, a method call must specify as many of the object's ancestors as are required to uniquely identify it.

The following table shows how fully qualified identifiers are constructed:

GUI Object	Fully-Qualified Identifier	Example
Main Window	The main window's identifier	<code>TextEdit.SetActive ()</code>
Dialog	The dialog's identifier	<code>Find.SetActive ()</code>
Control	The identifiers of the dialog and the control	<code>Find.Cancel.Click ()</code>

GUI Object	Fully-Qualified Identifier	Example
Menu item	The identifiers of the main window, the menu, and the menu item	<code>TextEditor.File.Open.Pick ()</code>

The fully qualified identifier for main windows and dialog boxes does not need to include ancestors because the declarations begin with the keyword `window`.

An identifier is the GUI object's logical name. By default, Silk Test Classic derives the identifier from the object's actual label or caption, removing any embedded spaces or special characters (such as accelerators). So, for example, the **Save As** label becomes the identifier **SaveAs**. Identifiers can contain single-byte international characters, such as `é` and `ñ`.

If the object does not have a label or caption, Silk Test Classic constructs an identifier by combining the class of the object with the object's index. When you are using the Classic Agent, the index is the object's order of appearance, from top left to bottom right, in relation to its sibling objects of the same class. For example, if a text box does not have a label or caption, and it is the first text box within its parent object, the default identifier is `TextField1`. When you are using the Open Agent, the index depends on the underlying technology of the application under test.



Note: The identifier is arbitrary, and you can change the generated one to the unique name of your choice.

Save the Test Frame

To save a test frame, click **File > Save** when the test frame is the active window. If it is a new file, it is automatically named `frame.inc`. If you already have a `frame.inc` file, a number is appended to the file name. You can click **File > Save** to select another name.

If you are working within a project, Silk Test Classic automatically adds the new test frame (`.inc`) to the project.

When saving a file, Silk Test Classic does the following:

- Saves a source file, giving it the `.inc` extension. The source file is an ASCII text file, which you can edit. For example: `myframe.inc`.
- Saves an object file, giving it the `.ino` extension. The object file is a binary file that is executable, but not readable by you. For example: `myframe.ino`.

Specifying How a Dialog Box is Invoked

4Test provides two equivalent ways to invoke a dialog box:

- Use the `Pick` method to pick the menu item that invokes the dialog box. For example:
`TextEditor.File.Open.Pick ()`
- Use the `Invoke` method: `Open.Invoke ()`

While both are equivalent, using the `Invoke` method makes your test cases more maintainable. For example, if the menu pick changes, you only have to change it in your window declarations, not in any of your test cases.

The Invoke method

To use the `Invoke` method, you should specify the `wInvoke` variable of the dialog box. The variable contains the identifier of the menu item or button that invokes the dialog box. For example:

```
window DialogBox Open
    tag "Open"
    parent TextEditor
WINDOW wInvoke = TextEditor.File.Open
```

Improving Object Recognition with Microsoft Accessibility

You can use Microsoft Accessibility (Accessibility) to ease the recognition of objects at the class level. There are several objects in Internet Explorer and in Microsoft applications that Silk Test Classic can better recognize if you enable Accessibility. For example, without enabling Accessibility Silk Test Classic records only basic information about the menu bar in Microsoft Word and the tabs that appear. However, with Accessibility enabled, Silk Test Classic fully recognizes those objects.

Example

Without using Accessibility, Silk Test Classic cannot fully recognize a `DirectUIHwnd` control, because there is no public information about this control. Internet Explorer uses two `DirectUIHwnd` controls, one of which is a popup at the bottom of the browser window. This popup usually shows the following:

- The dialog box asking if you want to make Internet Explorer your default browser.
- The download options **Open**, **Save**, and **Cancel**.

When you start a project in Silk Test Classic and record locators against the `DirectUIHwnd` popup, with accessibility disabled, you will see only a single control. If you enable Accessibility you will get full recognition of the `DirectUIHwnd` control.

Using Accessibility with the Open Agent

Win32 uses the Accessibility support for controls that are recognized as generic controls. When Win32 locates a control, it tries to get the accessible object along with all accessible children of the control.

Objects returned by Accessibility are either of the class `AccessibleControl`, `Button` or `CheckBox`. `Button` and `Checkbox` are treated specifically because they support the normal set of methods and properties defined for those classes. For all generic objects returned by Accessibility the class is `AccessibleControl`.

Example

If an application has the following control hierarchy before Accessibility is enabled:

- Control
 - Control
- Button

When Accessibility is enabled, the hierarchy changes to the following:

- Control
 - Control
 - Accessible Control
 - Accessible Control
 - Button
- Button

Enabling Accessibility for the Open Agent

If you are testing a Win32 application and Silk Test Classic cannot recognize objects, you should first enable Accessibility. Accessibility is designed to enhance object recognition at the class level.

To enable Accessibility for the Open Agent:

1. Click **Options > Agent**. The **Agent Options** dialog box opens.
2. Click **Advanced**.
3. Select the **Use Microsoft Accessibility** option. Accessibility is turned on.

Calling Windows DLLs from 4Test

This section describes how you can call Windows DLLs from 4Test.



Note: The Open Agent supports DLL calling for both 32-bit and 64-bit DLL calls, while the Classic Agent supports DLL calling only for 32-bit calls.

Silk Test Classic supports only the `_stdcall` calling convention.



Note: In some versions of Silk Test Classic, you can also use the `_cdecl` calling convention, although it is not officially supported. The `_cdecl` calling convention does not work with Silk Test 14.0 or later. Using the `_cdecl` calling convention might lead to unexpected failures of previously functioning DLL calls when migrating from the Classic Agent to the Open Agent or when upgrading Silk Test Classic to a newer version in which `_cdecl` does not work. If you are facing such failing DLL calls, ensure that you are using the `_stdcall` calling convention with the `_stdcall` naming decoration rules applied. For additional information on the DLL calling conventions, see [/Gd, /Gr, /Gv, /Gz \(Calling Convention\)](#).

Aliasing a DLL Name

If a DLL function has the same name as a 4Test reserved word, or the function does not have a name but an ordinal number, you need to rename the function within your 4Test declaration and use the 4Test alias statement to map the declared name to the actual name.

For example, the `exit` statement is reserved by the 4Test compiler. Therefore, to call a function named `exit`, you need to declare it with another name, and add an alias statement, as shown here:

```
dll "mydll.dll"
my_exit ()
alias "exit"
```

Calling a DLL from within a 4Test Script

A declaration for a DLL begins with the keyword `dll`. The general format is:

```
dll dllname.dll
prototype
[prototype]...
```

where `dllname` is the name of the dll file that contains the functions you want to call from your 4Test scripts and `prototype` is a function prototype of a DLL function you want to call.

Environment variables in the DLL path are automatically resolved. You do not have to use double backslashes (`\\`) in the code, single backslashes (`\`) are sufficient.

The Open Agent supports calling both 32bit and 64bit DLLs. You can specify which type of DLL the Open Agent should call by using the `SetDllCallPrecedence` method of the `AgentClass` class. If you do not know if the DLL is a 32bit DLL or a 64bit DLL, use the `GetDllCallPrecedence` function of the `AgentClass` Class. The Classic Agent provides support for calling 32bit DLLs only.

Silk Test Classic supports only the `_stdcall` calling convention.



Note: In some versions of Silk Test Classic, you can also use the `_cdecl` calling convention, although it is not officially supported. The `_cdecl` calling convention does not work with Silk Test 14.0 or later. Using the `_cdecl` calling convention might lead to unexpected failures of previously functioning DLL

calls when migrating from the Classic Agent to the Open Agent or when upgrading Silk Test Classic to a newer version in which `_cdecl` does not work. If you are facing such failing DLL calls, ensure that you are using the `_stdcall` calling convention with the `_stdcall` naming decoration rules applied. For additional information on the DLL calling conventions, see [/Gd, /Gr, /Gv, /Gz \(Calling Convention\)](#).

Prototype syntax

A function prototype has the following form:

```
return-type func-name ( [arg-list] )
```

where:

- return-type** The data type of the return value, if there is one.
- func-name** An identifier that specifies the name of the function.
- arg-list** A list of the arguments passed to the function, specified as follows:

```
[pass-mode] data-type identifier
```

where:

- pass-mode** Specifies whether the argument is passed into the function (`in`), passed out of the function (`out`), or both (`inout`). If omitted, `in` is the default.

To pass by value, make a function parameter an `in` parameter.

To pass by reference, use an `out` parameter if you only want to set the parameter's value; use an `inout` parameter if you want to get the parameter's value and have the function change the value and pass the new value out.
- data-type** The data type of the argument.
- identifier** The name of the argument.

You can call DLL functions from 4Test scripts, but you cannot call member functions in a DLL.

Example

The following example writes the text *hello world!* into a field by calling the `SendMessage` DLL function from the DLL `user32.dll`.

```
use "mswtype.inc"
use "mswmsg32.inc"

dll "user32.dll"
  inprocess ansicall INT SendMessage (HWND hWndParent, UINT
msg, WPARAM wParam, LPARAM lParam) alias "SendMessageA"

testcase SetTextViaDllCall()
  SendMessage(UntitledNotepad.TextField.GetHandle(),
WM_SETTEXT, 0, "hello world! ")
```

Passing Arguments to DLL Functions

Valid data types for arguments passed to DLL functions

Since DLL functions are written in C, the arguments you pass to these functions must have the appropriate C data types. In addition to the standard 4Test data types, Silk Test Classic also supports the following C data types:

- char, int, short, and long
- unsigned char, unsigned int, unsigned short, and unsigned long
- float and double



Note: Any argument you pass must have one of these data types (or be a record that contains fields of these types).

Passing string arguments

The `char*` data type in C is represented by the 4Test `STRING` data type. The default string size is 256 bytes.

The following code fragments show how a char array declared in a C struct is declared as a `STRING` variable in a 4Test record:

```
// C declaration
typedef struct
{
...
char szName[32];
...
}

// 4Test declaration
type REC is record
...
STRING sName, size=32
...
```

To pass a `NULL` pointer to a `STRING`, use the `NULL` keyword in 4Test. If a DLL sets an out parameter of type `char*` to a value larger than 256 bytes, you need to initialize it in your 4Test script before you pass it to the DLL function. This will guarantee that the DLL does not corrupt memory when it writes to the parameter. For example, to initialize an out parameter named `my_parameter`, include the following line of 4Test code before you pass `my_parameter` to a DLL:

```
my_parameter = space(1000)
```

If the user calls a DLL function with an output string buffer that is less than the minimum size of 256 characters, the original string buffer is resized to 256 characters and a warning is printed. This warning, `String buffer size was increased from x to 256 characters` (where `x` is the length of the given string plus one) alerts the user to a potential problem where the buffer used might be shorter than necessary.

Passing arguments to functions that expect pointers

When passing pointers to C functions, use these conventions:

- Pass a 4Test string variable to a DLL that requires a pointer to a character (null terminated).
- Pass a 4Test array or list of the appropriate type to a DLL that requires a pointer to a numerical array.
- Pass a 4Test record to a DLL that requires a pointer to a record. 4Test records are always passed by reference to a DLL.
- You cannot pass a pointer to a function to a DLL function.

Passing arguments that can be modified by the DLL function

An argument whose value will be modified by a DLL function needs to be declared using the `out` keyword. If an argument is sometimes modified and sometimes not modified, then declare the argument as `in` and then, in the actual call to the DLL, preface the argument with the `out` keyword, enclosed in brackets.

For example, the third argument (`lParam`) to the `SendMessage` DLL function can be either `in` or `out`. Therefore, it is declared as follows:

```
// the lParam argument is by default an in argument
dll "user.dll"
```



```
LRESULT  
SendMessage (HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
```

Then, to call the DLL with an out argument, you use the keyword `out`, enclosed within brackets:

```
SendMessage (Open.hWnd, WM_GETTEXT, 256, [out] sText)
```

Passing window handles to a DLL function

If a parameter takes a window handle, use the `hwnd` property or the `GetHandle` method of the `AnyWin` class to get the window handle you need.

Using DLL Support Files Installed with Silk Test Classic

Silk Test Classic is installed with the following include files that contain all the declarations, data types, and constants necessary for you to call hundreds of functions within the Windows API from your scripts.

msw32.inc Contains use statements for the include files that apply to 32-bit Windows: `mswconst.inc`, `mswtype.inc`, `mswfun32.inc`, `mswmsg32.inc`, and `mswutil.inc`.

By including `msw32.inc` in your 4Test scripts, you have access to all the information in the other include files.



Note: The DLL functions declared in the files included in `msw32.inc` are aliased to the W (wide-character) functions.

mswconst.inc Declares constants you pass to DLL functions. These constants contain style bits, message box flags, codes used by the `GetSystemMetrics` function, flags used by the `GetWindow` function, window field offsets for the `GetWindowLong` and the `GetWindowWord` functions, class field offsets for the `GetClassLong` and `GetClassWord` functions, and menu function flags.

mswfun32.inc Contains 4Test declarations for 32-bit functions in the `user32.dll` and `kernel32.dll` files. The `mswfun32.inc` file provides wide character support. This means that you no longer have to edit `mswfun32.inc` in order to call Windows DLL functions. See the description of `mswfun32.inc` in the Dll declaration section.

mswmsg32.inc Declares 32-bit Microsoft Window messages, control messages, and notification codes.

mswtype.inc Declares many data types commonly used in the Windows API.

mswutil.inc Contains the following utility functions:

- `PrintWindowDetail`
- `GetStyleBitList`
- `PrintStyleBits`

Extending the Class Hierarchy


This section describes how you can extend the class hierarchy.

Classes

This section describes the 4Test classes.

Overview of Classes

The `class` indicates the type, or kind, of GUI object being declared.

 **Note:** This is the `4Test` class, not the class that the GUI itself uses internally. For example, although the class might be `Label` on one GUI and `Text` on another, `4Test` uses the class name `StaticText` to refer to text strings that cannot be edited.

A class defines data and behavior

The class also defines methods (actions) and properties (data) that are inherited by the GUI object. For example, if you record a declaration for a pushbutton named `OK`, a test case can legally use a method like `Click` on the pushbutton because the `Click` method is defined at the class level. In other words, the definition of what it means to click on a pushbutton is included within the definition of the `4Test` class itself, and this definition is inherited by each pushbutton in the GUI. If this were not true, you would have to define within each GUI object's window declaration all the methods you wanted to use on that object.

The class as recorded cannot be changed

The one exception is that if the recorded class is `CustomWin`, meaning that Silk Test Classic does not recognize the object. You can, when appropriate, map the class to one that is recognized.

Custom classes

Enable an application to perform functions specific to the application and to enhance standard class functionality. Custom classes are also easy to maintain and can be extended easily by developers. All custom objects default to the built-in class, `CustomWin`.

Custom objects fall into two general categories:

- | | |
|--------------------------|---|
| Visible objects | Objects that Silk Test Classic knows about, but cannot identify, for example, the icon in an About dialog box. Two further categories of visible objects include: <ul style="list-style-type: none">• Common objects are those that look and behave like standard objects, for example, a third-party object that looks and acts like a <code>PushButton</code>, but is recorded as a <code>CustomWin</code>.• Uncommon objects, on the other hand, have no relation to the existing standard objects. For example, an <code>Icon</code>. there is no corresponding <code>Icon</code> class. |
| Invisible objects | Objects that Silk Test Classic cannot recognize at all. |

Polymorphism

If a class defines its own version of a method or property, that method or property overrides the one inherited from an ancestor. This is referred to as polymorphism. For example, the `ListBox` class has its own `GetContents` method, which overrides the `GetContents` method inherited from the `AnyWin` class.

CursorClass, ClipboardClass, and AgentClass

The following three classes are not part of the `AnyWin` class hierarchy, because they define methods for objects that are not windows:

- | | |
|-----------------------|--|
| CursorClass | Defines the three methods you can use on the cursor: <code>GetPosition</code> , <code>GetType</code> , and <code>Wait</code> . |
| ClipboardClass | Defines the two methods you can use on the system clipboard: <code>GetText</code> and <code>SetText</code> . |

AgentClass Defines the methods you can use to set options in the 4Test Agent. The 4Test Agent is the component of Silk Test Classic that translates the method calls in your test cases into the appropriate GUI- specific event streams.

Predefined identifiers for Cursor, Clipboard, and Agent

You do not record declarations for the cursor, the clipboard, or the Agent. Instead, you use predefined identifiers for each of these objects when you want to use a method to act against the object. The predefined methods for each are:

- 4Test Agent: `Agent`
- clipboard: `Clipboard`
- cursor (mouse pointer): `Cursor`

For example, to set a 4Test Agent option, you use a call such as the following:

```
Agent.SetOption (OPT_VERIFY_COORD, TRUE)
```

Defining New Classes with the Open Agent

This functionality is supported only if you are using the Open Agent.

Consider the declarations for the **Open** and the **Save As** dialog boxes of the **Text Editor** application, which each contain exactly the same child windows:

window DialogBox Open

```
locator "Open"  
parent TextEditor  
StaticText FileNameText  
    locator "File Name:"  
TextField FileName1  
    locator "File Name:"  
ListBox FileName2  
    locator "File Name:"  
StaticText DirectoriesText  
    locator "Directories:"  
StaticText PathText  
    locator "#3"  
ListBox Path  
    locator "#2"  
StaticText ListFilesOfTypeText  
    locator "List Files of Type:"  
PopupList ListFilesOfType  
    locator "List Files of Type:"  
StaticText DrivesText  
    locator "Drives:"  
PopupList Drives  
    locator "Drives:"  
PushButton OK  
    locator "OK"  
PushButton Cancel  
    locator "Cancel"  
PushButton Network  
    locator "Network"
```

window DialogBox SaveAs

```
locator "Save As"  
parent TextEditor  
StaticText FileNameText  
    locator "File Name:"  
TextField FileName1
```

```

locator "File Name:"
ListBox FileName2
locator "File Name:"
StaticText DirectoriesText
locator "Directories:"
StaticText PathText
locator "#3"
ListBox Path
locator "#2"
StaticText ListFilesOfTypeText
locator "List Files of Type:"
PopupMenu ListFilesOfType
locator "List Files of Type:"
StaticText DrivesText
locator "Drives:"
PopupMenu Drives
locator "Drives:"
PushButton OK
locator "OK"
PushButton Cancel
locator "Cancel"
PushButton Network
locator "Network"

```

It is not uncommon for an application to have multiple dialogs whose only difference is the caption: The child windows are all identical or nearly identical. Rather than recording declarations that repeat the same child objects, it is cleaner to create a new class that groups the common child objects.

For example, here is the class declaration for a new class called `FileDialog`, which is derived from the `DialogBox` class and declares each of the children that will be inherited by the **SaveAs** and **Open** dialog boxes:

```

winclass FileDialog : DialogBox
parent TextEditor
StaticText FileNameText
locator "File Name:"
TextField FileName1
locator "File Name:"
ListBox FileName2
locator "File Name:"
StaticText DirectoriesText
locator "Directories:"
StaticText PathText
locator "#3"
ListBox Path
locator "#2"
StaticText ListFilesOfTypeText
locator "List Files of Type:"
PopupMenu ListFilesOfType
locator "List Files of Type:"
StaticText DrivesText
locator "Drives:"
PopupMenu Drives
locator "Drives:"
PushButton OK
locator "OK"
PushButton Cancel
locator "Cancel"
PushButton Network
locator "Network"

```

To make use of this new class, you must do the following:

1. Rewrite the declarations for the **Open** and **Save As** dialog boxes, changing the class to **FileDialog**.

2. Remove the declarations for the child objects inherited from the new class.

Here are the rewritten declarations for the **Open** and **Save As** dialog boxes:

```
window FileDialog SaveAs
    locator "Save As"
window FileDialog Open
    locator "Open"
```

For more information on the syntax used in declaring new classes, see the `winclass` declaration.

The default behavior of Silk Test Classic is to tag all instances of the parent class as the new class. So, if you record a window declaration against a standard object from which you have defined a new class, Silk Test Classic records that standard object's class as the new class. To have all instances declared by default as the original class, add the following statement to the declaration of your new class: `setting DontInheritClassTag = TRUE`. For example, let's say you define a new class called `FileDialog` and derive it from the `DialogBox` class. Then you record a window declaration against a dialog box. Silk Test Classic records the dialog box to be of the new `FileDialog` class, instead of the `DialogBox` class. To have Silk Test Classic declare the class of the dialog box as `DialogBox`, in the `FileDialog` definition, set `DontInheritClassTag` to `TRUE`. For example:

```
winclass FileDialog : DialogBox
    setting DontInheritClassTag = TRUE
```

Defining New Class Properties

You can define new properties for existing classes using the `property` declaration. You use these class properties to hold data about an object; you can use class properties anywhere in a script.

DesktopWin

Because the desktop is a GUI object, it derives from the `AnyWin` class. However, unlike other GUI objects, you do not have to record a declaration for the desktop. Instead, you use the predefined identifier `Desktop` when you want to use a method on the desktop.

For example, to call the `GetActive` method on the desktop, you use a call like the following:

```
wActive = Desktop.GetActive ()
```

Logical Classes

The `AnyWin`, `Control`, and `MoveableWin` classes are logical (virtual) classes that do not correspond to any actual GUI objects, but instead define methods common to the classes that derive from them. This means that Silk Test Classic never records a declaration that has one of these classes.

Furthermore, you cannot extend or override logical classes. If you try to extend a logical class, by adding a method, property or data member to it, that method, property, or data member is not inherited by classes derived from the class. You will get a compilation error saying that the method/property/data member is not defined for the window that tries to call it. Nor can you override the class, by rewriting existing methods, properties, or data members. Your modifications are not inherited by classes derived from the class.

Class Hierarchy (Open Agent)

You can define your own methods and properties, as well as define your own classes. You can also define your own attributes, which are used in the verification stage in test cases.

The `4Test` class hierarchy defines the methods and properties that enable you to query, manipulate, and verify the data or state of any GUI object in your application. You can define your own methods and properties, as well as define your own classes. You can also define your own attributes, which are used in the verification stage in test cases. The following schema shows a listing of the built-in class hierarchy for the core classes and the Open Agent:

- AgentClass
- AnyWin
 - Control
 - CheckBox
 - ComboBox
 - Group
 - Link
 - ListBox
 - ListViewEx
 - PageList
 - PushButton
 - RadioList
 - Scale
 - StaticText
 - StatusBar
 - TableEx
 - TextField
 - ToggleButton
 - ToolBar
 - TreeView
 - DesktopWinOA
 - Item
 - SeparatorItem
 - ToolItem
 - CheckBoxToolItem
 - DropDownToolItem
 - RadioListToolItem
 - PushToolItem
 - ScrollBar
 - HorizontalScrollBar
 - VerticalScrollBar
 - TableColumn
 - TableRow
 - Menu
 - MenuItem
 - MoveableWin
 - DialogBox
 - MainWin
 - WinPart
- ClipboardClass
- ConsoleClass
- CursorClass

Verifying Attributes and Properties

This section describes how you can use attributes and properties to verify test cases.

Attribute Definition and Verification

When you record a test case, you can verify the test case using attributes.

You can choose to verify using either attributes or properties. Generally you will verify using properties, because property verification is more flexible.

For example, the attributes for the `DialogBox` class are `Caption`, `Contents`, `Default button`, `Enabled`, and `Focus`. The following 4Test code implements the `Default Button` attribute in the `winclass.inc` file:

```
attribute "Default button", VerifyDefaultButton, QueryDefaultButton
```

As this 4Test code shows, each attribute definition begins with the statement, followed by the following three comma-delimited values:

1. The text that you want to display in the **Attribute** panel of the **Verify Window** dialog box. This text must be a string.
2. The method Silk Test Classic should use to verify the value of the attribute at runtime.
3. The method Silk Test Classic should use to get the actual value of the attribute at runtime.

Defining a New Attribute for an Existing Class

To add one or more attributes to an existing class, use the following syntax:

```
winclass ExistingClass : ExistingClass...  
attribute_definitions
```

Each attribute definition begins with the `attribute` statement, followed by the following three comma-delimited values:

1. The text that you want to display in the **Attribute** panel of the **Verify Window** dialog box. This text must be a string.
2. The method Silk Test Classic should use to verify the value of the attribute at runtime.
3. The method Silk Test Classic should use to get the actual value of the attribute at runtime.

Each attribute definition must begin and end on its own line. When you define a new attribute, you usually need to define two new methods (steps 2 and 3 above) if none of the built-in methods suffice.

Silk Test Classic allows you to add, delete, or edit the existing functionality of a class; this applies to both functions and variables of a class. However, we recommend that you do not override a function or a variable by declaring a function or variable of that same name. Furthermore, you should never override a variable that has a tag associated with it. You cannot have two variables with the same name in the same level of an object. If you do so, Silk Test Classic will display a compile error.

Defining New Verification Properties

You can perform verifications in your test cases using properties. These verification properties are different from class properties, which are defined using the property declaration. Verification properties are used only when verifying the state of your application in a test case. Silk Test Classic comes with built-in verification properties for all classes of GUI objects.

You can define your own verification properties, which will be added to the built-in properties listed in the **Verify Window** dialog box when you record a test case.

Syntax for Attributes

To add one or more attributes to an existing class, use the following syntax:

```
winclass ExistingClass : ExistingClass...  
attribute_definitions
```

Each attribute definition must begin and end on its own line.

When you define a new attribute, you usually need to define two new methods if none of the built-in methods suffices.

For example, to add a new attribute to the `DialogBox` class that verifies the number of children in the dialog box, you add code like this to your test frame (or other include file):

```
winclass DialogBox:DialogBox

    attribute "Number of children", VerifyNumChild, GetNumChild

    integer GetNumChild()
        return ListCount (GetChildren ()) // return count of children of dialog

    hidecalls VerifyNumChild (integer iExpectedNum)
        Verify (GetNumChild (), iExpectedNum, "Child number test")
```

As this example shows, you use the `hidecalls` keyword when defining the verification method for the new attribute.

Hidecalls Keyword

The keyword `hidecalls` hides the method from the call stack listed in the results. Using `hidecalls` allows you to update the expected value of the verification method from the results. If you do not use `hidecalls` in a verification method, the results file will point to the frame file, where the method is defined, instead of to the script. We recommend that you use `hidecalls` in all verification methods so that you can update the expected values.

An Alternative to NumChildren as a Class Property

Instead of defining `NumChildren` as a class property, you could also define it as a variable, then initialize the variable in a script. For example, in your include file, you would have:

```
winclass DialogBox : DialogBox
INTEGER NumChild2
// list of custom verification properties
LIST OF STRING lsPropertyName = {"NumChild2"}
```

And in your script, before you do the verification, you would initialize the value for the dialog box under test, such as:

```
Find.NumChild2 = ListCount(Find.GetChildren ())
```

Defining Methods and Custom Properties

This section describes how you can define methods and custom verification properties.

Defining a New Method

To add a method to an existing class, you use the following syntax to begin the method definition:

```
winclass ExistingClass : ExistingClass
```

The syntax `ExistingClass : ExistingClass` means that the declaration that follows extends the existing class definition, instead of replacing it.



Note: Adding a method to an existing class adds the method to all instances of the class.

Example

To add a `SelectAll()` method to the `TextField` class, add the following code to your `frame.inc` file:

```
winclass TextField : TextField
    SelectAll()
    TypeKeys("<Ctrl+a>")
```

In your test cases, you can then use the `SelectAll` method like any other method in the `TextField` class.

```
UntitledNotepad.TextField.SelectAll()
```

Defining a New Method for a Single GUI Object

To define a new method to use on a single GUI object, not for an entire class of objects, you add the method definition to the window declaration for the individual object, not to the class. The syntax is exactly the same as when you define a method for a class.

To add a method to a single GUI object, for example to add the `SelectAll()` method to a specific `TextField` object, locate the GUI object in your `frame.inc` file, like described in the following code sample:

```
window MainWin UntitledNotepad
...
TextField TextField
    locator "//TextField"
```

In your test cases, you can then use the `SelectAll` method like any other method of the `TextField` object:

```
window MainWin UntitledNotepad
...
TextField TextField
    locator "//TextField"
    SelectAll()
    TypeKeys("<Ctrl+a>")
```



Note: Adding a method to a single GUI object adds the method only to the specific GUI object and not to other instances of the class.

Classic Agent Example

For example, suppose you want to create a method named `SetLineNum` for a dialog box named **GotoLine**, which performs the following actions:

- Invokes the dialog box.
- Enters a line number.
- Clicks **OK**.

The following 4Test code shows how to add the definition for the `SetLineNum` method to the declaration of the **GotoLine** dialog box.

```
window DialogBox GotoLine
    tag "Goto Line"
    parent TextEditor
    const wInvoke = TextEditor.Search.GotoLine

    void SetLineNum (STRING sLine)
```

```
Invoke () // open dialog
Line.SetText (sLine) // populate text field
// whose identifier is Line
Accept () // close dialog, accept values

//Then, to go to line 7 in the dialog, you use this method
call in your testcases:
GotoLine.SetLineNum (7)
```

Recording a Method for a GUI Object

If you need to perform an action on an object, which is not supported by the class of the object, you can record or manually script a new method that performs the action. Add the new method to the class or to the window declaration of the object to use the method in the same way as the built-in methods of the class.

You can use the **Record Method** dialog box to record a method for a class or window declaration.



Note: Before you can record a method, you must have recorded window declarations.

1. Open an include file or script file.
2. Move the mouse cursor into the declaration of the GUI object to which you want to add a method.
3. Click **Record > Method**.



Note: This is only available when an include file or script file is the active window and the mouse cursor is within a class or object declaration because methods are necessarily part of a class or object definition.

The **Record Method** dialog box appears.

4. Type a name for the method into the **Method name** field.

You can also select one of the predefined methods:

- BaseState
- Close
- Invoke
- Dismiss

5. Click **Start Recording**. Silk Test Classic is minimized and displays the application under test.
 - If you are using the Open Agent, the **Recording** window appears.
 - If you are using the Classic Agent, the **Record Status on Classic Agent** dialog box appears.
6. Perform and record the actions that you require.
7. Stop recording.
 - If you are using the Open Agent, click **Stop Recording**.
 - If you are using the Classic Agent, click **Done**.

The **Method code** field in the **Record Method** dialog box displays the actions that you have recorded translated into 4Test statements.

8. On the **Record Method** dialog box, click **OK** to paste the code into the file in the editor.
9. Edit the 4Test statements that were recorded, if necessary.



Note: To add a method to a class which is using the Open Agent, you can also copy the method into the script from a recorded test case.

Deriving a New Method from an Existing One

To derive a new method from an existing method, you can use the derived keyword followed by the scope resolution operator (::).

Use the following syntax:

```
new method : existing method
```

The following example defines a `GetCaption` method for `WPFNewTextBox` that prints the string `Caption` as `is` before calling the built-in `GetCaption` method (defined in the `AnyWin` class) and printing its return value:

```
winclass WPFNewTextBox : WPFTextBox
GetCaption ()
Print ("Caption as is: ")
Print (derived::GetCaption ())
```

Defining Custom Verification Properties

1. In a class declaration or in the declaration for an individual object, define the variable `lsPropertyNamees` as follows:

```
LIST OF STRING lsPropertyNamees
```

2. Specify each of your custom verification properties as elements of the list `lsPropertyNamees`. Custom verification properties can be either:
 - Class properties, defined using the property statement.
 - Variables of the class or individual object.

Any properties you define in `lsPropertyNamees` will override built-in properties with the same name. With your custom verification properties listed as elements in `lsPropertyNamees`, when you record and run a test case, those additional properties will be available during verification.

Redefining a Method

There may be some instances in which you want to redefine an existing method. For example, to redefine the `GetCaption` method of the `Anywin` class, you use this 4Test code:

```
winclass AnyWin : AnyWin
GetCaption ()
// insert method definition here
```

Confirming the Property List

You can use the `GetPropertyList` method to confirm the list of verification properties for an object. For example, the following simple test case prints the list of all the verification properties of the **Find** dialog to the results file:

```
testcase FindDialogPropertyConfirm ()
TextEditor.Search.Find.Pick ()
ListPrint (Find.GetPropertyList ())
Find.Cancel.Click ()
```

Examples

This section provides examples for defining methods and custom verification properties.

Example: Adding a Method to TextField Class

This example adds to the `TextField` class a method that selects all of the text in the text box.

```
winclass TextField : TextField
SelectAll ()
STRING sKey1, sKey2
switch (GetGUIType ())
case mswnt, msw2003
sKey1 = "<Ctrl-Home>"
sKey2 = "<Shift-Ctrl-End>"
```

```

        case mswvista
            sKey1 = "<Ctrl-Up>"
            sKey2 = "<Shift-Cmd-Down>"
        // return cursor to 1,1
        this.TypeKeys (sKey1)
        // highlight all text
        this.TypeKeys (sKey2)
    
```

The keyword `this` refers to the object the method is being called on.

The preceding method first decides which keys to press, based on the GUI. It then presses the key that brings the cursor to the beginning of the field. It next presses the key that highlights (selects) all the text in the field.

Example: Adding Tab Method to DialogBox Class

To add a `Tab` method to the `DialogBox` class, you could add the following 4Test code to your `frame.inc` file (or other include file):

```

winclass DialogBox : DialogBox
Tab (INTEGER iTimes optional)
if (iTimes == NULL)
    iTimes = 1
this.TypeKeys ("<tab {iTimes}>")
    
```

Example: Defining a Custom Verification Property

Let's look at an example of defining a custom verification property. Say you want to test a dialog box. Dialog boxes come with the following built-in verification properties:

- Caption
- Children
- DefaultButton
- Enabled
- Focus
- Rect
- State

And let's say that you have defined a class property, `NumChildren`, that you want to make available to the verification system.

Here is the class property definition:

```

property NumChildren
INTEGER Get ()
return ListCount (GetChildren ())
    
```

That property returns the number of children in the object, as follows:

- The built-in method `GetChildren` returns the children in the dialog box in a list.
- The built-in function `ListCount` returns the number of elements in the list returned by `GetChildren`.

To make the `NumChildren` class property available to the verification system (that is, to also make it a verification property) you list it as an element in the variable `lsPropertyNamees`. So here is part of the extended `DialogBox` declaration that you would define in an include file:

```

winclass DialogBox : DialogBox
// user-defined property
property NumChildren
    INTEGER Get ()
        return ListCount (GetChildren ())
// list of custom verification properties
LIST OF STRING lsPropertyNamees = {"NumChildren"}
    
```

Now when you verify a dialog box in a test case, you can verify your custom property since it will display in the list of DialogBox properties to verify.



Note: As an alternative, instead of defining NumChildren as a class property, you could also define it as a variable, then initialize the variable in a script. For example, in your include file, you would have:

```
winclass DialogBox : DialogBox
    INTEGER NumChild2
    // list of custom verification properties
    LIST OF STRING lsPropertyName = {"NumChild2"}
```

And in your script-before you do the verification-you would initialize the value for the dialog box under test, such as:

```
Find.NumChild2 = ListCount (Find.GetChildren ())
```

Porting Tests to Other GUIs

This section describes how you can port tests to other GUIs.

Handling Differences Among GUIs

This section describes how you can handle differences between GUIs when porting tests to other GUIs.

Conditionally Loading Include Files

If you are testing different versions of an application, such as versions that run on different platforms or versions in different languages, you probably have different include files for the different versions. For example, if your applications run under different languages, you might have text strings that display in windows defined in different include files, one per language. You want Silk Test Classic to load the proper include file for the version of the application you are currently testing.

Load Different Include Files for Different Versions of the Test Application

1. Define a compiler constant.
For example, you might define a constant named `MyIncludeFile`.
2. Insert the following statement into your 4Test file: `use constant`.
For example, if you defined a constant `MyIncludeFile`, insert the following statement: `use MyIncludeFile`. In this example, constant can also be an expression that evaluates to a constant at compile time.
3. When you are ready to compile your 4Test files, specify the file name of the include file you want loaded as the value of the constant in the **Compiler Constants** dialog box.
Be sure to enclose the value in quotation marks if it is a string.
4. Compile your code.

Silk Test Classic evaluates all compiler constants and substitutes their values for the constants in your code. In this case, the constant `MyIncludeFile` will be evaluated to a file, which will be loaded through the `use` statement.

Different Error Messages

The `VerifyErrorBox` function, shown below, illustrates how to solve the problem of different error messages on each GUI platform. For example, if a GUI platform always adds the prefix "Error:" to its

message, while the other platforms do not, you might use or create a GUI Specifier for that platform and then use the `VerifyErrorBox` function as follows:

```
VerifyErrorBox (STRING sMsg)
    // verifies that the error box has the correct error
    // message, then dismisses the error box

    const ERROR_PREFIX = "ERROR: "
    const ERROR_PREFIX_LEN = Len (ERROR_PREFIX)
    STRING sActMsg = MessageBox.Message.GetText ()

    // strip prefix "ERROR: " from GUI Specifier for that platform error
messages
    if (GetGUIType () == GUI Specifier for that platform)
        sActMsg = SubStr (sActMsg, ERROR_PREFIX_LEN + 1)

    Verify (sActMsg, sMsg)
    MessageBox.Accept ()
```

One Logical Control can Have Two Implementations

Consider the case where the same logical control in your application is implemented using different classes on different GUIs.

If the kinds of actions you can perform against the object classes are similar, and if Silk Test Classic uses the same method names for the actions, then you do not have a portability problem to address.

For example, the methods for the `RadioList` and `PopupList` classes have identical names, because the actions being performed by the methods are similar. Therefore, if a control in your application is a popup list on one GUI and a radio list on another, your scripts are already portable.

If the two object classes do not have similar methods, or if the methods have different names, then you need to port your scripts.

Options Sets and Porting

Options sets save all current options except General Options. Options sets can be very useful when trying to use the same scripts on different operating systems. The primary differences between the two may be compiler constants.

For example, you might use the compiler constant `sCmdLine`. Usually, the command line to invoke an application differs between the PC operating systems. You could create a compiler constant (note that there is a string limit on compiler constants) for use in the `sCmdLine` constant to differentiate between the platforms' command lines. You might also use a compiler constant for methods that work slightly differently on the two operating systems, such as the `Pick()` methods.

Specifying Options Sets

In a test plan, you can specify options sets to be used with the test plan or parts of it. You use options sets to automatically run different tests that require different options without having to manually open options sets.

To ensure that everyone working on a project has the same options settings (such as class mapping), do one of the following:

- Open an Options Set.
- Set these option values at runtime.
- Specify the following statement in the test plan: `optionset: filename.opt.`

Dependent test cases will run with the specified options set opened. The options set will be closed when it passes out of scope. If you don't specify a full path name, the file is considered to be in a directory relative to the directory containing the current test plan or sub-plan.

Remember:

- Options can also be set at runtime in a test script by using the `Agent` method, `SetOption`, and passing in the name of the option and its value.
- Many Agent options and their values are found in the **Agent Options** dialog box.
- Agent options can be set in a `testcase/` function.
- Class map settings, set at runtime, are best set before any tests are executed (for example, in `ScriptEnter`) and after each test case (for example `TestcaseExit`) in case any have been changed in the course of a test case.
- Class mappings set at runtime using the Agent method `SetOption` are only in effect during test execution; these settings are not available to the recorders.

Supporting Differences in Application Behavior

Although you can account for differences in the appearance of your application in the window declarations, if the application's behavior is fundamentally different when ported, you need to modify your test cases themselves. To modify your test cases, you write sections of 4Test code that are platform-specific, and then branch to the correct section of code using the return value from the `GetGUIType` built-in function.

This topic shows how to use the `GetGUIType` function in conjunction with `if` statements and the `switch` statements.

Switch statements

You can use GUI specifiers before an entire `switch` statement and before individual statements within a `case` clause, but you cannot use GUI specifiers before entire `case` clauses.

```
testcase GUISwitchExample()
INTEGER i
FOR i=1 to 5
mswxp, mswnt switch(i)

// legal:
mswxp, mswnt switch (i)
  case 1
    mswxp Print ("hello")
    mswnt Print ("goodbye")
  case 2
    mswxp raise 1, "error"
    mswnt Print ("continue")
  default
    mswxp Print ("ok")

// NOT legal:
switch (i)
  mswxp case 1
    Print ("hello")
  mswnt case 1
    Print ("goodbye")
```

If statements

You can use GUI specifiers in `if` statements, as long as GUI specifiers used within the statement are subsets of any GUI specifiers that enclose the entire `if` statement.

```
// legal because no GUI specifier
// enclosing entire if statement:
if (i==j)
  msw32, mswnt Print ("hi")
  msw2000 Print ("bye")

// legal because msw is a subset of enclosing specifier:
msw32, msw2000 if (i==j)
  mswnt Print("hi")
```

```

// legal for the same reason as preceding example:
msw32, msw2000 if (i==j)
    Print ("hi")
mswnt else
    Print ("Not the same")

// NOT legal because msw2000 is not a subset
// of the enclosing GUI specifier msw:
msw32 if (i==j)
    msw2000 Print ("bye") // Invalid GUI type

```

If you are trying to test multiple conditions, then you should use a `select` or `switch` block. You could use nested `if..else` statements, but if you have more than two or three conditions, the levels of indentation will become cumbersome.

You should not use an `if..else if..else` block. Although `if..else if..else` will work, it will be difficult to troubleshoot exceptions that occur because the results file will always point to the first `if` statement even if it was actually a subsequent `if` statement that raised the exception.

For example, in the following test case, the third string, `Not a date`, will raise the exception:

```
*** Error: Incompatible types -- 'Not a date' is not a valid date
```

The exception actually occurs in the lines containing:

```
GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
```

For the nested `if..else` and the `select` blocks, the results file points to those lines as the sources of the exceptions. However, for the `if..else if..else` block, the results file points to the first `if` statement, in other words to the line:

```
[ - ] if IsNull (sVal)
```

even though that line clearly is not the source of the exception because it does not concern `DATETIME` values.

```

[+] testcase IfElseIfElse ()
[-] LIST OF STRING lsVals = {...}
[ ] "2006-05-20"
[ ] "2006-11-07"
[ ] "Not a date"
[ ] STRING sVal
[ ]
[-] for each sVal in lsVals
[-] do
[-] if IsNull (sVal)
[ ] Print ("No date given")
[-] else if sVal == FormatDateTime (GetDateTime (), "yyyy-mm-dd")
[ ] Print ("The date is today")
[-] else if GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
[ ] Print ("The year is this year")
[-] else
[ ] Print ("Some other year")
[-] except
[ ] ExceptLog ()
[ ]
[-] do
[-] if IsNull (sVal)
[ ] Print ("No date given")
[-] else
[-] if sVal == FormatDateTime (GetDateTime (), "yyyy-mm-dd")
[ ] Print ("The date is today")
[-] else
[-] if GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
[ ] Print ("The year is this year")

```



```

[-] else
[ ] Print ("Some other year")
[-] except
[ ] ExceptLog ()
[ ]
[-] do
[-] select
[-] case IsNull (sVal)
[ ] Print ("No date given")
[-] case sVal == FormatDateTime (GetDateTime (), "yyyy-mm-dd")
[ ] Print ("The date is today")
[-] case GetDateTimePart ([DATETIME]sVal, DTP_YEAR) == 2006
[ ] Print ("The year is this year")
[-] default
[ ] Print ("Some other year")
[-] except
[ ] ExceptLog ()
[ ]

```

Text Box Requires Return Keystroke

On some GUIs, the `Enter/Return` key must be pressed after data is entered into a text box. Suppose you want to create a test case that enters invalid data into the text box, and then checks if the application detects the error. After the test case enters the invalid data, it needs to use the `GetGUIType` function to determine the GUI, and then press the `Return` key if the GUI requires it.

For example:

```

// code to enter an invalid string into field
if (GetGUIType () == mswnt)
    MyTextField.TypeKeys ("<Return>")
// code to verify that application detected error

```

Using Cross-Platform Methods in Your Scripts

In scripts, you can use your cross-platform method names. The window declarations map the cross-platform method names you use in your scripts to the actual methods required to carry out the actions you want on each of the GUIs.

Continuing the example from *Creating a Class that Maps to Several Silk Test Classic Classes*, you use the `Select` method in your code to select the control named `Direction`.

```

testcase SearchBackward ()

    LISTITEM Item
    Item = "Up"
    Find.Invoke ()
    Find.Direction.Select (Item)
    .
    .
    .
    Find.Dismiss ()

```



Note: The script does not indicate that anything unusual is happening. All of the steps necessary to make the `Select` method work properly, regardless of the class of the object, are encapsulated in the class and window declarations.

About GUI Specifiers

This section describes GUI specifiers.

Class Declarations

Be careful using GUI specifiers before class declarations; they can be ambiguous. Any ambiguities must be resolvable at compile-time.

```
// bad style:
msw winclass myclass
mswnt winclass myclass
window myclass inst // Ambiguous. Is it an instance of
                    // the msw class or the mswnt class?
```

The preceding example's ambiguity can be resolved by specifying a GUI target with conditional compilation (so that, for example, only code for msw gets compiled, in which case inst would be an instance of the msw class or by explicitly using a GUI specifier for the window, as follows:

```
// good style:
msw winclass myclass
mswnt winclass myclass
msw window myclass inst
```

Conditional Compilation

If you have GUI-specific code in your scripts and declarations, you can have Silk Test Classic conditionally compile your code based on the values of the GUI specifiers - only code specific to a particular GUI is compiled (as well, of course, as all code that is not GUI-specific). This has the following two advantages:

- The compilation is faster.
- The resulting code is smaller and requires less memory to run.

You can also cause conditional compilation by using constants, which are evaluated at compile time.

Constants are not restricted to conditional compilation. You can use constants for any value that you want resolved at compile time.

Conditionally Compile Code

1. Prefix any 4Test statements that are GUI-specific with the appropriate GUI specifier.
2. Specify the platforms that you want to compile for by entering the appropriate GUI specifiers in the **GUI Targets** field in the **Runtime Options** dialog box. You can specify as many GUI targets as you want; separate each GUI specifier by a comma.
Setting a GUI target affects which classes are listed in the **Library Browser**.
3. To conditionalize code based on the value of constants you define, do the following:
 1. Click **Compiler Constants** in the **Runtime Options** dialog box.
 2. The **Compiler Constants** dialog box is displayed.
 3. Define a constant and specify its value.
 4. Use the constant in your code anywhere you can specify an expression.
4. Click **OK** to close the **Runtime Options** dialog box.

GUI with Inheritance

When using GUI specifiers for parent classes, you must explicitly use the GUI specifiers with the descendants:

```
mswxp winclass newclass
mswxp winclass subclass : newclass
mswxp window subclass inst
```

GUI with Global Variables

Be careful when using GUI specifiers with global variables, because Silk Test Classic initializes global variables before connecting to an Agent. This might not give you the results you want if you are doing distributed testing.

Let's say that you are running tests on a remote machine that is listed in the **Runtime Options** dialog box. Because Silk Test Classic initializes all global variables before connecting to an Agent, any GUI specifier at the global level will initialize to the host machine, not the target machine you want to test against.

For example, say the host machine is running a different operating system than the target machine. Consider the following script:

```
mwxp STRING sVar1 = SYS_GetEnv("UserName")

mwxp STRING sVar1 = SYS_GetRegistryValue
    (HKEY_LOCAL_MACHINE, "System\CurrentControlSet\Control", "Current
User")

main()
    print(sVar1)
```

This script fails, with the error message:

```
*** Error: Registry entry 'Current User' not found
```

because `sVar1` is initialized to the value for the host system, not the target system.

Constants behave similarly to global variables if you use a GUI specifier to initialize the variable (or constant). It is a good idea to use GUI specifiers in the main function, under **Options > Runtime** or another function that is called after the Agent is contacted.

Marking 4Test Code as GUI Specific

Using Silk Test Classic, you can create portable test cases that will test your application on any of the supported GUIs. The reason for this is that your test cases use logical names, called identifiers, to refer to the GUI objects, and not actual names, called tags. Therefore, if there are differences in the ported application's appearance, you need only change the window declarations, not the test cases themselves.

The porting scenarios described section use 4Test keywords called GUI specifiers to indicate that portions of include files or script files are specific to a particular GUI. Before studying these scenarios, you should understand which GUI specifiers are available and how to use them in your include files and script files.

4Test includes a long list of GUI specifiers.

Syntax of a GUI Specifier

A GUI specifier has this syntax:

```
[[gui-type [,gui-type]] | [!gui-type]]
```

`gui-type` is the GUI. You can express this in one of two mutually exclusive ways. For example, you can specify one or more GUIs separated by commas, as in:

```
mwxp, mswin7
```

Or you can specify all but one GUI, as in the following, which indicates that what follows applies to all environments except Windows NT-based operating systems:

```
! mswnt
```

What Happens when the Code is Compiled

Only code relevant to the GUI environments specified in the GUI Targets field (plus all common code) will be compiled. If you do not list any GUI specifiers in the GUI Targets field, all code will be compiled; at runtime, code not relevant to the platform the application is running on will be skipped.

The constants you have defined are evaluated and used to compile the code. You can use this feature to conditionally load include files.

Where You Use GUI Specifiers

A GUI specifier can be located before any 4Test declaration or statement except the `use` statement, which must be evaluated at compile time, with the following exceptions:

- `Switch` statements
- `If` statements
- `Type` statements
- `Do... except` statements
- `Class` declarations
- `GUI` with inheritance
- `GUI` with global variables

If you try to use a browser specifier instead of a GUI specifier to specify a window, Silk Test Classic will generate an error. The primary use of browser specifiers is to address differences in window declarations between different browsers. Each Agent connection maintains its own browser type, allowing different threads to interact with different browsers.

do...except Statements

You can use GUI specifiers to enclose an entire `do . . . except` statement before individual statements, but you cannot use GUI specifiers before the `except` clause.

```
// legal:
do
    mswxp Verify (expr1,expr2)
    mswin7 Verify (expr3,expr4)
except
    mswin7 reraise
    mswxp if (ExceptNum () == 1)
        Print ("err, etc.")
// NOT legal:
mswin7 do
    Verify (expr,expr)
mswxc except
    reraise
```

Type Statements

You can use a GUI specifier before a `type type . . . is enum` or `type . . . is set` statement, but not before an individual value within the type declaration.

Supporting GUI-Specific Objects

This section describes how Silk Test Classic supports testing GUI-specific objects.

Supporting GUI-Specific Captions

Classic Agent

When you are using the Classic Agent, by default Silk Test Classic bases the tag for an object on the actual caption or label of the object. If the captions or labels change when the application is ported to a different GUI, you have two options:

- You can have multiple tags, each based on the platform-specific caption or label.
- You can have a single tag, using the index form of the tag, as long the relative position of the object is the same in the ported versions of the application.

Then, in your test cases, you can use the same identifier to refer to the object regardless of what the object's actual label or caption is.

Open Agent

When you are using the Open Agent, Silk Test Classic creates locator keywords in an INC file to create scripts that use dynamic object recognition and window declarations. The locator is the actual name of the object, as opposed to the identifier, which is the logical name. Silk Test Classic uses the locator to identify objects in the application when executing test cases. Test cases never use the locator to refer to an object; they always use the identifier.

The advantages of using locators with an INC file include:

- You combine the advantages of INC files with the advantages of dynamic object recognition. For example, scripts can use window names in the same manner as traditional, Silk Test Classic tag-based scripts and leverage the power of XPath queries.
- Enhancing legacy INC files with locators facilitates a smooth transition from using hierarchical object recognition to new scripts that use dynamic object recognition. You use dynamic object recognition but your scripts look and feel like traditional, Silk Test Classic tag-based scripts that use hierarchical object recognition.
- You can use `AutoComplete` to assist in script creation. `AutoComplete` requires an INC file.

Supporting GUI-Specific Executables

The command to start the application will almost always be different on each GUI. The `Invoke` method of Silk Test Classic expects to find the command in the constant `sCmdLine`, which is defined in the main window declaration of your application. You should declare as many `sCmdLine` variables as there are GUIs on which your application runs, beginning each declaration with the appropriate GUI specifier.

For example, the following constants specify how Silk Test Classic should start the Text Editor application on Windows and Windows Vista:

```
m32 const sCmdLine = "c:\program files\<SilkTest install directory>\silktest\textedit.exe"
m32 const sCmdLine = "{SYS_GetEnv('SEGUE_APPS')}/SilkTest/demo/textedit"
```

Supporting GUI-Specific Menu Hierarchies

When an application is ported, there are two common structural differences in the menu hierarchy:

- The menu bar contains a platform-specific menu.
- A menu contains different menu items.

To illustrate the case of the platform-specific menu, consider the Microsoft Windows system menu or a Vista menu (for example). Silk Test Classic recognizes these kinds of standard GUI-specific menus and includes the appropriate GUI specifier for them when you record declarations.

For menus that Silk Test Classic does not recognize as platform-specific, you should preface the window declaration with the appropriate GUI specifier.

Different menu items - example

To illustrate the case of different menu items, suppose that the **Edit** menu for the Text Editor application has a menu item named **Clear** which displays on the Windows version only. The declaration for the **Edit** menu should look like the following:

Classic Agent	Open Agent
<pre>Menu Edit tag "Edit" msw32 MenuItem Clear tag "Clear" MenuItem Undo tag "Undo"</pre>	<pre>Menu Edit locator "Edit" msw32 MenuItem Clear locator "Clear" MenuItem Undo locator "Undo"</pre>

Supporting Custom Controls

This sections describes how Silk Test Classic supports custom controls.

Why Silk Test Classic Sees Controls as Custom Controls

A control is defined by the following:

- The actual class name of the control.
- The underlying software code that creates and manipulates the control.

Whenever the definition of a control varies from the standard, Silk Test Classic defines the control as a custom control. During recording, Silk Test Classic attempts to identify the class of each control in your GUI and to assign the appropriate class from the built-in class hierarchy. If a control does not correspond to one of the built-in classes, Silk Test Classic designates the control as a custom control.

- When you are using the Classic Agent, Silk Test Classic assigns custom controls to the `CustomWin` class.
- When you are using the Open Agent, Silk Test Classic assigns custom controls to the `Control` class or another class.

Classic Agent Example

For example, Silk Test Classic supports the standard MFC library, which is a library of functions that allow for the creation of controls and the mechanism of interaction with them. In supporting these libraries, Silk Test Classic contains algorithms to interrogate the controls based upon the standard libraries. When these algorithms do not work, Silk Test Classic reports the control as a `CustomWin`.

Suppose that you see a text box in a window in your application under test. It looks like a normal text field, but Silk Test Classic calls it a control of the class `CustomWin`.

Reasons Why Silk Test Classic Sees the Control as a Custom Control

For the following reasons Silk Test Classic might recognize a control as a custom control:

- The control is not named with the standard name upon the definition of the control in the application under test. For example, when a **TextField** is named **EnterTextRegion**. If this is the only reason why

Silk Test Classic recognizes the control as a custom control, then you can class map the control to the standard name.

The class mapping might not work. The class mapping will work if the control is not really a custom control, but rather a standard control with a non-standard name. Try this as your first attempt at dealing with a custom control.

- If the class mapping does not work the control truly is a custom control. The software in the application under test that creates and manipulates the control is not from the standard library. That means that the Silk Test Classic algorithms written to interrogate this kind of control will not work, and other approaches will have to be used to manipulate the control.

When you are using the Classic Agent, the support for custom controls depends on whether the control is a graphical control, such as a tool bar, or a non-graphical control, such as a text box.

Supporting Graphical Controls

If an application contains a graphical area, for example a tool bar, which is actually composed of a discrete number of graphical controls, Silk Test Classic records a single declaration for the entire graphical area; it does not understand that the area contains individual controls.

Custom Controls (Open Agent)

This functionality is supported only if you are using the Open Agent.

Silk Test Classic provides the following features to support you when you are working with custom controls:

- The *dynamic invoke* functionality of Silk Test Classic enables you to directly call methods, retrieve properties, or set properties on an actual instance of a control in the application under test (AUT).
- The *class mapping* functionality enables you to map the name of a custom control class to the name of a standard Silk Test class. You can then use the functionality that is supported for the standard Silk Test class in your test.

Silk Test Classic supports managing custom controls over the UI for the following technology domains:

- Win32
- Windows Presentation Foundation (WPF)
- Windows Forms
- Java AWT/Swing
- Java SWT
- The **Manage Custom Controls** dialog box enables you to specify a name for a custom control that can be used in a locator and also enables you to write reusable code for the interaction with the custom control.



Note: For custom controls, you can only record methods like `Click`, `TextClick`, and `TypeKeys` with Silk Test Classic. You cannot record custom methods for custom controls except when you are testing Apache Flex applications.

Dynamic Invoke

Dynamic invoke enables you to directly call methods, retrieve properties, or set properties, on an actual instance of a control in the application under test. You can also call methods and properties that are not available in the Silk Test Classic API for this control. Dynamic invoke is especially useful when you are working with custom controls, where the required functionality for interacting with the control is not exposed through the Silk Test Classic API.

Call dynamic methods on objects with the `DynamicInvoke` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Call multiple dynamic methods on objects with the `DynamicInvokeMethods` method. To retrieve a list of supported dynamic methods for a control, use the `GetDynamicMethodList` method.

Retrieve dynamic properties with the `GetProperty` method and set dynamic properties with the `SetProperty` method. To retrieve a list of supported dynamic properties for a control, use the `GetPropertyList` method.

For example, to call a method named `SetTitle`, which requires the title to be set as an input parameter of type string, on an actual instance of a control in the application under test, type the following:

```
control.DynamicInvoke("SetTitle", {"my new title"})
```



Note: Typically, most properties are read-only and cannot be set.



Note: Reflection is used in most technology domains to call methods and retrieve properties.



Note: You cannot dynamically invoke methods for DOM elements.

Frequently Asked Questions About Dynamic Invoke

This functionality is supported only if you are using the Open Agent.

This section includes a collection of questions that you might encounter when you are dynamically invoking methods to test custom controls.

Which Methods Can I Call With the DynamicInvoke Method?

This functionality is supported only if you are using the Open Agent.

To get a list of all the methods that you can call with the `DynamicInvoke` method for a specific test object, you can use the `GetDynamicMethodList`. To view the list, you can for example print it to the console or view it in the debugger.

Why Does an Invoke Call Return a Simple String when the Expected Return is a Complex Object?

This functionality is supported only if you are using the Open Agent.

The `DynamicInvoke` method can only return simple data types. Complex types are returned as string. Silk Test Classic uses the `ToString` method to retrieve the string representation of the return value. To call the individual methods and read properties of the complex object that is returned by the first method invocation, use `DynamicInvokeMethods` instead of `DynamicInvoke`.

How Can I Simplify My Scripts When I Use Many Calls To DynamicInvokeMethods?

This functionality is supported only if you are using the Open Agent.

When you extensively use `DynamicInvokeMethods` in your scripts, the scripts might become complex because you have to pass all method names as strings and all parameters as lists. To simplify such complex scripts, create a static method that interacts with the actual control in the AUT instead of interacting with the control through `DynamicInvokeMethods`.

Testing Apache Flex Custom Controls

Silk Test Classic supports testing Apache Flex custom controls. However, by default, Silk Test Classic cannot record and playback the individual sub-controls of the custom control.

For testing custom controls, the following options exist:

- Basic support

With basic support, you use dynamic invoke to interact with the custom control during replay. Use this low-effort approach when you want to access properties and methods of the custom control in the test application that Silk Test Classic does not expose. The developer of the custom control can also add

methods and properties to the custom control specifically for making the control easier to test. A user can then call those methods or properties using the dynamic invoke feature.

The advantages of basic support include:

- Dynamic invoke requires no code changes in the test application.
- Using dynamic invoke is sufficient for most testing needs.

The disadvantages of basic support include:

- No specific class name is included in the locator, for example Silk Test Classic records `//FlexBox` rather than `//FlexSpinner`.
- Only limited recording support.
- Silk Test Classic cannot replay events.

For more details about dynamic invoke, including an example, see *Dynamically Invoking Apache Flex Methods*.

- **Advanced support**

With advanced support, you create specific automation support for the custom control. This additional automation support provides recording support and more powerful play-back support. The advantages of advanced support include:

- High-level recording and playback support, including the recording and replaying of events.
- Silk Test Classic treats the custom control exactly the same as any other built-in Apache Flex control.
- Seamless integration into Silk Test Classic API
- Silk Test Classic uses the specific class name in the locator, for example Silk Test Classic records `//FlexSpinner`.

The disadvantages of advanced support include:

- Implementation effort is required. The test application must be modified and the Open Agent must be extended.

Managing Custom Controls (Open Agent)

This functionality is supported only if you are using the Open Agent.

You can create custom classes for custom controls for which Silk Test Classic does not offer any dedicated support. Creating custom classes offers the following advantages:

- Better locators for scripts.
- An easy way to write reusable code for the interaction with the custom control.

Example: Testing the `tabControl` Infragistics control

Suppose that a custom tab control is recognized by Silk Test Classic as the generic class `Control`. Using the custom control support of Silk Test Classic has the following advantages:

Better object recognition because the custom control class name can be used in a locator.

Many objects might be recognized as `Control`. The locator requires an index to identify the specific object. For example, the object might be identified by the locator `//Control[13]`. When you create a custom class for this control, for example the class `UltraTabControl`, you can use the locator `//UltraTabControl`. By creating the custom class, you do not require the high index, which would be a fragile object identifier if the application under test changed.

You can implement reusable playback actions for the control in scripts.

When you are using custom classes, you can encapsulate the behavior for getting the contents of a grid into a method by adding the following code to your custom class, which is the class that gets generated when you specify the custom control in the user interface.

Typically, you can implement the methods in a custom control class in one of the following ways:

- You can use methods like `Click`, `TypeKeys`, `TextClick`, and `TextCapture`. In this example the `TextClick` method is used.
- You can dynamically invoke methods on the object in the AUT.

Without using the custom classes, when you want to select a tab in your custom tab controls, you can write code like the following:

```
UltraTabControl.TextClick( "<TabName>
")
```

When you are using custom classes, you can encapsulate the behavior for selecting a tab into a method by adding the following code to your custom class, which is the class that gets generated when you specify the custom control in the user interface:

```
void SelectTab(string tabText)
    TextClick(tabText)
```

The custom class looks like the following:

```
winclass UltraTabControl : Control
    tag "[UltraTabControl]"

    void SelectTab(string tabText)
        TextClick(tabText)
```

You can now use the newly created method `SelectTab` in a script like the following:

```
UltraTabControl.SelectTab( "<TabName>
")
```

When you define a class as a custom control, you can use the class in the same way in which you can use any built-in class, for example the `Dialog` class.

Supporting a Custom Control

This functionality is supported only if you are using the Open Agent.

Silk Test Classic supports managing custom controls over the UI for the following technology domains:

- Win32
- Windows Presentation Foundation (WPF)
- Windows Forms


- Java AWT/Swing
- Java SWT


To create a custom class for a custom control for which Silk Test Classic does not offer any dedicated support.


1. Click **Options > Manage Custom Controls**. The **Manage Custom Controls** dialog box opens.
2. In the **Frame file for custom class declarations** field, type in a name or click **Browse** to select the frame file that will contain the custom control.
3. Click on the tab of the technology domain for which you want to create a new custom class.
4. Click **Add**.
5. Click one of the following:
 - Click **Identify new custom control** to directly select a custom control in your application with the **Identify Object** dialog box.
 - Click **Add new custom control** to manually add a custom control to the list.

A new row is added to the list of custom controls.

6. If you have chosen to manually add a custom control to the list:
 - a) In the **Silk Test base class** column, select an existing base class from which your class will derive. This class should be the closest match to your type of custom control.
 - b) In the **Silk Test class** column, enter the name to use to refer to the class. This is what will be seen in locators. For example: `//UltraGrid` instead of `//Control[13]`.

 **Note:** After you add a valid class, it will become available in the **Silk Test base class** list. You can then reuse it as a base class.
 - c) In the **Custom control class name** column, enter the fully qualified class name of the class that is being mapped. For example: `Infragistics.Win.UltraWinGrid.UltraGrid`. For Win32 applications, you can use the wildcards `?` and `*` in the class name.
7. *Only for Win32 applications:* In the **Use class declaration** column, set the value to **False** to simply map the name of a custom control class to the name of a standard Silk Test class. When you map the custom control class to the standard Silk Test class, you can use the functionality supported for the standard Silk Test class in your test. Set the value to **True** to additionally use the class declaration of the custom control class.
8. Click **OK**.
9. *Only for scripts:*
 - a) Add custom methods and properties to your class for the custom control.
 - b) Use the custom methods and properties of your new class in your script.

 **Note:** The custom methods and properties are not recorded.

 **Note:** Do not rename the custom class or the base class in the script file. Changing the generated classes in the script might result in unexpected behavior. Use the script only to add properties and methods to your custom classes. Use the **Manage Custom Controls** dialog box to make any other changes to the custom classes.

Custom Controls Dialog Box

This functionality is supported only if you are using the Open Agent.

Options > Manage Custom Controls.

Silk Test Classic supports managing custom controls over the UI for the following technology domains:

- Win32
- Windows Presentation Foundation (WPF)
- Windows Forms
- Java AWT/Swing
- Java SWT

In the **Frame file for custom class declarations**, define the frame file into which the new custom classes should be generated.

When you map a custom control class to a standard Silk Test class, you can use the functionality supported for the standard Silk Test class in your test. The following **Custom Controls** options are available:

Option	Description
Silk Test base class	Select an existing base class to use that your class will derive from. This class should be the closest match to your type of custom control.
Silk Test class	Enter the name to use to refer to the class. This is what will be seen in locators.
Custom control class name	Enter the fully qualified class name of the class that is being mapped. You can use the wildcards ? and * in the class name.
Use class declaration	This option is available only for Win32 applications. By default <code>False</code> , which means the name of the custom control class is mapped to the name of the standard Silk Test class. Set this setting to <code>True</code> to additionally use the class declaration of the custom control class.



Note: After you add a valid class, it will become available in the **Silk Test base class** list. You can then reuse it as a base class.

Example: Setting the options for the UltraGrid Infragistics control

To support the `UltraGrid Infragistics` control, use the following values:

Option	Value
Silk Test base class	<code>Control</code>
Silk Test class	<code>UltraGrid</code>
Custom control class name	<code>Infragistics.Win.UltraWi nGrid.UltraGrid</code>

Using Clipboard Methods

If you are having trouble getting or setting information with a custom object that contains text, you might want to try the 4Test Clipboard methods. For example, assume you have a class, `CustomTextBuffer`, which is similar to a `TextField`, but using the `GetText` and `SetText` methods of the `TextField` does not work with the `CustomTextBuffer`. In such a case, you can use the `GetText` and `SetText` methods of the `ClipboardClass`.

Get and Set Text Sample Code

The following sample code retrieves the contents of the `CustomTextBuffer` by placing it on the **Clipboard**, then printing the **Clipboard** contents:

```
// Go to beginning of text field
CustomTextBuffer.TypeKeys ("<Ctrl-Home>")
// Highlight it
CustomTextBuffer.TypeKeys ("<Ctrl-Shift-End>")
```

```
// Copy it to the Clipboard
CustomTextBuffer.TypeKeys (<Ctrl-Insert>)
// Print the contents of the Clipboard
Print (Clipboard.GetText())
```

Setting text

Similarly, the following sample code inserts text into the custom object by pasting it from the Clipboard:

```
// Go to beginning of text field
CustomTextBuffer.TypeKeys (<Ctrl-Home>)
// Highlight it
CustomTextBuffer.TypeKeys (<Ctrl-Shift-End>)
// Paste the Clipboard contents into the text field
CustomTextBuffer.TypeKeys (<Shift-Insert>)
```

You can wrap this functionality in `GetText` and `SetText` methods you define for your custom class, similar to what was shown in supporting custom text boxes.

Using the Modified Declaration

Once you create window declarations like these for the graphical objects in your application, you can manipulate them as you would any other object. For example, if the tool bar was contained in an application named `MyApp`, to click on the **FileOpen** icon in the tool bar, you use the following command:

```
MyApp.FileOpen.Click()
```

You need to write this statement, and others that access the objects declared above, such as `Save` and `Printer`, by hand. **Record > Testcase** and **Record > Actions** will not use these identifiers.

Filtering Custom Classes

This section describes how you can filter custom classes.

Invisible Containers

Sometimes a window contains an invisible dialog box that contains controls. You can set these "dialog box containers" to Ignore using class mapping and style-bits in order to avoid making all of the dialog boxes disappear.

See the following examples for details.

Example: WordPad with No Class Mappings

```
[ - ] window MainWin WordPad
[ + ] multitag "*"WordPad"
[ + ] Menu File
[ + ] Menu Edit
[ + ] Menu View
[ + ] Menu Insert
[ + ] Menu Format
[ + ] Menu Help
// toolbars seen, but are nested
[ + ] CustomWin BottomStatusBar
[ - ] CustomWin Frame
[ + ] CustomWin FormatBar
[ + ] ComboBox ComboBox1
[ + ] ComboBox ComboBox2
[ + ] CustomWin StandardBar
[ + ] CustomWin Ruler
[ - ] main ()
WordPad.Frame.FormatBar.ComboBox1.Select ("Arial")
```

Example: WordPad with AfxControlBar Ignored

```
[ - ] window MainWin WordPad
[ + ] multitag " *WordPad"
[ + ] Menu File
[ + ] Menu Edit
[ + ] Menu View
[ + ] Menu Insert
[ + ] Menu Format
[ + ] Menu Help
// toolbars, ruler, and statusbar not seen
[ + ] ComboBox ComboBox1
[ + ] ComboBox ComboBox2
[ + ] TextField Document
[ - ] main ()
WordPad.ComboBox1.Select ( "Arial" )
```

Supporting Internationalized Objects

This section describes how you can work with internationalized objects.

Overview of Silk Test Classic Support of Unicode Content

Silk Test Classic is Unicode-enabled, which means that Silk Test Classic is able to recognize double-byte (wide) languages. We have enabled components within the application to deal with Unicode content. The Silk Test Classic GUI supports the display and input of wide text. The 4Test language processor has been enhanced to support wide text. All 4Test library functions have been widened. The extensions have been enhanced to support the input and output of wide text.

We have added and modified 4Test functions to deal with internationalization issues. With Silk Test Classic you can test applications that contain content in double-byte languages such as Chinese, Korean, or Japanese (Kanji) characters, or any combination of these. You can also name Silk Test Classic files using internationalized characters. Silk Test Classic supports three text file formats: ANSI, Unicode and UTF-8.

Silk Test Classic supports the following:

- Localized versions of Windows.
- International keyboards and native language Input Method Editors (IME).
- Passing international strings as parameters to test cases, methods, and so on, and comparing strings.
- Accessing databases through direct ODBC standard access.
- Reading and writing text files in multiple formats: ANSI, Unicode, and UTF-8.

For information about new features, supported platforms, and tested versions, refer to the [Release Notes](#).

Before testing double-byte characters with Silk Test Classic

Testing an internationalized application, particularly one that contains double-byte characters, is more complicated than testing an application that contains strictly English single-byte characters. Testing an internationalized application requires that you understand a variety of issues, from operating system support, to language packs, to fonts, to working with IMEs and complex languages.

Before you begin testing your application using Silk Test Classic, you must do the following:

- Meet the needs of your application under test (AUT) for any necessary localized OS, regional settings, and required language packs.
- Install the fonts necessary to display your AUT.

- If you are testing an application that requires an IME for data input, install the appropriate IME.

Using DB Tester with Unicode Content

To use DB Tester with Unicode characters:

- You must have a Unicode-capable driver (ODBC version 3.5 or higher) associated with the data source name you are using in your test plan.
- The database must be Unicode capable (SQL Server 7 and 2000, Oracle 8 and higher).

Issues Displaying Double-Byte Characters

When you are dealing with internationalized content, being able to display the content of your application is critical. Carefully consider the following:

Operating system	Your operating system needs to be capable of displaying double-byte characters in the system dialog boxes and menus used by your application.
Silk Test Classic	You need to be concerned about displaying your content in the Silk Test Editor and the Silk Test Classic dialog boxes.
Application under test	You need to have a font installed that is capable of displaying the content of your application. If you have multiple languages represented in your application, you will need a font that spans these languages.
Browser	If your application is web-based, make sure that you are using a browser that supports your content, that the browser is configured to display your content, and that you have the necessary fonts installed to display your application.
Complex scripts (languages)	Silk Test Classic does not support complex scripts such as the bi-directional languages Hebrew and Arabic. These are languages that require special processing to display and edit because the characters are not laid out in a simple linear progression from left to right, as are most western European characters.

Learning More About Internationalization

There are a variety of online sites that provide general information about internationalization issues. You may find the following Web sites useful if you are learning about internationalization, localization or Unicode. They include:

- Microsoft's Professional Developer's Site for Software Globalization Information (<http://www.microsoft.com/globaldev/default.asp>)
- The definitive word on the W3C's Web site (<http://www.w3.org/international>)
- The Unicode Consortium, a non-profit organization founded to develop, extend and promote use of the Unicode Standard (<http://www.unicode.org>)
- IBM's International Components for Unicode (<http://oss.software.ibm.com/icu/userguide/index.html>)
- A tutorial from Sun on how to internationalize Java applications (<http://java.sun.com/docs/books/tutorial/i18n>)

Silk Test Classic File Formats

Silk Test Classic gives you the ability to specify the file format of text files and .ini files. Before Silk Test Classic 5.5, all files were in the ANSI file format. You can create the following formats:

ANSI	For Silk Test Classic purposes, ANSI is defined as the Microsoft Windows ANSI character set from Code Page 1252 Windows Latin 1.
-------------	--

Unicode Is an extended form of ASCII that provides the ability to represent data in a uniform plaintext format that can be sorted and processed efficiently. Unicode encompasses nearly all characters used in computers today.

UTF-8 Unicode Transformation Format (UTF) Is a multi-byte encoding that can handle all Unicode characters. It is used to compress Unicode, minimize storage consumption and maximize transmission.

You have the ability to save text files in any of three file formats: ANSI, UTF-8, and Unicode. By default all files are saved in UTF-8 format. The **Save As** dialog boxes throughout include a list box from which you can select the file format in which you want to save your file.

- ANSI files cannot contain non ANSI characters
- The file formats available will depend on the content of your text file. If your file contains characters not available on code page 1252, ANSI will not display in the list box. If you are working with an existing ANSI file and add non-ANSI characters, the **Save As** dialog box will open when you attempt to save the file. In order to save the changes you will need to change the file format and click **Save**.
- The title bar indicates the file format: When you have a file open, the format of that file is indicated on the title bar.
- Silk Test Classic uses the Microsoft standard Byte Order Marked (BOM) to determine the file type for UTF-8 and Unicode files. If a Unicode file does not have the BOM marker then Silk Test Classic sees the file as an ANSI file, and the Unicode characters cannot be displayed.

Reusing Silk Test Classic Single-Byte Files as Double-Byte

If you have existing single-byte Silk Test Classic text files, such as *.pln, *.inc or *.t, that you want to use in double-byte testing, the files must:

- Be compatible with Silk Test Classic, such as files created using the IE 5.x DOM extension for testing a Web application.
- Be recompiled in Silk Test Classic because the object files, *.ino and *.to, are not compatible.

Opening an existing Silk Test Classic file as a double-byte file

Choose one of the following:

- Copy the file you want to re-use to a new directory. Do not copy the associated object (*.ino or *.to) files. In Silk Test Classic, open this new file.
- In the existing directory, delete the object files associated with the file you want to re-use. In Silk Test Classic, open the desired file.

When the Silk Test Classic file is compiled, new objects files are created. If you enter double-byte content into the file, when you try to close the file you will be prompted to save the file in a compatible file format, Unicode or UTF-8.

Specifying File Formats for Existing Files with Unicode Content

If you want to save an existing file in a different file format, choose one of the following:

Overwriting the file

If the file is already referenced from other files, you may want to change the format without changing the name or its location. As you cannot have two files with the same name saved in the same directory, even in different formats, the only option is to overwrite the file.

1. Make sure the file is the active window. Click **File > Save As** and select the file from the list.
2. From the **Save as format** list box, select the file format. ANSI is not available if the file contains characters outside of the ANSI character set.
3. Click **Save**. A dialog box displays asking if you want to overwrite the file.

4. Click **Yes**.

Saving in the same directory

If you want to have versions of a file in various formats within the same directory, you must save each file with a different name.

1. Make sure the file is the active window. Click **File > Save As**.
2. In the **File name** text box, enter the new name of the file.
3. From the **Save as format** list box, select the file format. ANSI is not available if the file contains characters outside of the ANSI character set.
4. Click **Save**.

Saving in a different directory

If you would like to keep the name of the file but change the format, you must save the file in a different directory.

1. Make sure the file is the active window. Click **File > Save As** and select the file from the list.
2. Navigate to the directory in which you want to save the file.
3. From the **Save as format** list box, select the file format. ANSI is not available if the file contains characters outside of the ANSI character set.
4. Click **Save**.

If you modify an ANSI text file and the modifications include characters outside of the ANSI characters set, when you try to save your changes, the Save As dialog box will open and you need to either overwrite the ANSI file with a file of the same name but in a different format, or rename the file and save in Unicode or UTF-8 format .

Specifying File Formats for New Files with Unicode content

This topic contains instructions on specifying the file format for:

With the exception of test frames, to specify the file format of a new file:

1. Click **File > New**.
2. On the **New** dialog box, select the file type.
3. Click **OK**. The untitled file opens.
4. Click **File > Save As**. The **Save As** dialog box opens.
5. Navigate to where you want to store the file and enter the name of the file in the **File name** text box.
6. Select a file format (UTF-8 is the default) from the **Save as format** list box. ANSI is not available if the file contains characters outside of the ANSI character set.
7. Click **Save**.

To specify the file format for a new test frame:

1. Click **File > New**.
2. On the **New** dialog box, select the file type **Test Frame** and click **OK**. The **New Test Frame** dialog box opens.
3. To select a file format, click **Browse**. The **Save As** dialog box opens. The default file format for test frames is UTF-8. If you simply type the path and file name in the **File name** text box of the **New Test Frame** dialog box and click **OK**, the file is saved in UTF-8.
4. Navigate to where you want to store the file and enter the name the file in the **File name** text box.
5. Select the file format from the **Save as format** list box. If you select ANSI and if the file contains characters outside of the ANSI character set, when you try to save the file you will need to change the file format to a compatible format, Unicode or UTF-8.
6. Click **Save**. The **New Test Frame** dialog box regains focus.

7. On the **New Test Frame** dialog box, select the application and proceed as normal.

If you modify an ANSI text file and the modifications include characters outside of the ANSI characters set, when you try to save your changes, the **Save As** dialog box will open and you need to either overwrite the ANSI file with a file of the same name but in a different format, or rename the file and save in Unicode or UTF-8 format .

Working with Bi-Directional Languages

Silk Test Classic supports bi-directional languages to the extent that the operating system does. Silk Test Classic captures static text in all Unicode languages. However, scripting, playback and many string functions are not fully supported for complex languages, the most common of these being the bi-directional languages Hebrew and Arabic. The problems you may encounter are discussed below.

Silk Test Classic with bi-directional languages on Windows XP

Windows XP is a multi-lingual operating system and is capable of handling bi-directional languages when configured properly.

On Windows XP if you input characters from RIGHT to LEFT (CBA) provided that the default system locale is set for a bi-directional language, Silk Test Classic will correctly record and playback the characters as they were entered and display, from RIGHT to LEFT. When you use a 4Test string function such as `STRPOS` (string position) to return the third element, 4Test correctly counts from right to left and returns "C"

Once you have set a default system locale, the operating system continues to be able to read and write that language properly, even after another locale has been set as the default. This works only if the language is not unchecked from the **Language Settings** area after another default is set. Once a language is unchecked, the ability to read and write in that language will be gone when you reboot your system. You would need to reset it as the default to restore the capability.

Configuring Your Environment

This section describes how you can configure your environment for internationalized objects.

Configuring Your Microsoft Windows XP PC for Unicode Content

If you have already configured your Windows XP PC to run your internationalized application, you may be able to disregard this topic and see *Recording Identifiers for International Applications*.

On Microsoft Windows XP you may need to do all or some of the following:

- Install language support required by your application through modifications in the **Regional and Language Options** dialog box.
- If your application contains content that is in a large-character-set language, such as simplified Chinese, you may need to install an Input Method Editor (IME) if you want to input data in this language. For additional information about IMEs, refer to the Microsoft support site.

Fonts

To display the content of your application in Silk Test Classic you will need to have an appropriate font installed and specify this font in the system registry and in the Silk Test Classic Options/Editor Font.

Installing Language Support

You must have administrator privileges to install language packs or set the system default locale.

Microsoft Windows XP provides built-in support for many double-byte languages. Enabling this support can be done at the time of install or after setup through the **Regional and Language Options** dialog box. If you enable language support after setup, you may need files from the Microsoft Windows XP installation

CD. Configurations will vary depending on your needs and how your system has been configured previously. The following instructions are intended only to be general information to get you started:

1. Click **Start > Settings > Control Panel > Regional and Language Options**.
2. If you are testing East Asian languages, select the **Languages** tab, and then check the **Install files for East Asian languages** check box.
You may be prompted to insert the Microsoft Windows XP CD for the necessary files.
3. Click the **Advanced** tab on the **Regional and Language Options** dialog box.
4. Select the language that matches the language of the non-Unicode programs you want to use.
For example Chinese (PRC).
5. Click **OK**.
6. Reboot your computer for the changes to take effect.
After you restart your computer, if you want to input data in a language other than the default language, you must click the Language bar icon in your system tray and select the language from the multi-lingual indicator.

Setting Up Your Input Method Editor

If you want to use an Input Method Editor (IME) to input data in the language you selected, you may need to set up your IME.

1. Click **Start > Settings > Control Panel > Regional and Language Options**.
2. Click the **Languages** tab.
3. Click **Details** in the **Text Services and Input Language** area.
4. In the **Settings** tab on the **Text Services and Input Language** dialog box, select the language you want to use as your default input language.
5. In the **Preferences** section of the **Settings** tab, click **Language Bar**, make sure the **Show the Language Bar on the desktop** check box is checked, and then click **OK** on the **Settings** tab.
This default will enable your system to display this language in dialog boxes and menus. We recommend setting the default to the language of the AUT.

Displaying Double-Byte Characters

While Silk Test Classic can process Unicode, displaying double-byte characters is not automatic. Keep the following in mind:

- Is your operating system configured to display your content?
- Is Silk Test Classic configured to display double-byte content in its dialog boxes?
- Do you have the right font set to display your content in the Editor?

Displaying Double-Byte Characters in Dialog Boxes

If Silk Test Classic is rendering squares or pipes in dialog boxes where you expect double-byte characters, you may need to make a simple modification to Silk Test Classic using a script we have provided. This script is located in `<SilkTest Installation directory>\Tools`.

1. In Silk Test Classic, click **File > Open**.
2. In the `Tools` directory, open `font.t`.
3. Click **Run > Testcase**. The **Run Testcase** dialog box opens.
4. In the **arguments** area, type the name of the font in quotes.
For example, `Arial Unicode MS`. It is not necessary to include the type of font, for example `Arial Unicode MS (True Type)`.
5. Click **Run**.

6. Reboot your computer for the changes to take effect.

Displaying Double-Byte Characters in the Editor

In order for the Editor to display double-byte characters, such as those captured in your test frame, you must select a font that is able to display these characters.

1. In Silk Test Classic, click **Options > Editor Font**.
2. From the available fonts, select one that is able to display the language of your application.

If your application contains multiple languages, make sure that you have a font installed that is capable of rendering all the languages, as the Editor does not display multiple fonts. Licensed Microsoft Office 2000 users can freely download the Arial Unicode MS font from Microsoft.

Using an IME with Silk Test Classic

Silk Test Classic supports IMEs. The IME is enabled only after you have installed an Asian language package. The IME will work once you have installed it, enabled it, and are in an application with IME support. In Silk Test Classic, the IME is only available when a file, such as an include or script, is active.

For additional information about IMEs and for downloads, see the Microsoft support site.

Troubleshooting Unicode Content

This section contains topics to help troubleshoot unicode content.

Display Issues

This section describes how you can troubleshoot display issues in Unicode contents.

Why Are My Window Declarations Recording Only Pipes?

If your window declarations record only pipes (|), You've probably forgotten to set the **Options > Font Editor** to a font that can display the language of your AUT.

What Are Pipes and Squares Anyway?

The pipes and squares, or even question marks (?), display in place of characters which the system has not yet been configured to display. A font that does not support the language is being used in the dialog boxes and menus. Whether or not you see pipes or squares depends on what font is used and what language you are trying to display.

Why Can I Only Enter Pipes Into a Silk Test Classic File?

If you can only enter pipes into a file, for example a frame file or an include file, the Silk Test Classic Editor font is not set to display the language of your AUT.

Why Do I See Pipes and Squares in the Project Tab?

Pipes, squares, and questions marks (?) display in place of characters which the system has not yet been configured to display. A font that does not support the language is being used in the dialog boxes and menus. Whether or not you see pipes or squares depends on what font is used and what language you are trying to display.

You must configure your system and make sure that you have set the regional settings.

Why Cannot My System Dialog Boxes Display Multiple Languages?

If you are testing an application whose content contains multiple languages, meaning that it has several character sets represented, you may need to:

- Make sure that you have a font installed on your machine that can display all the languages.
- Configure Silk Test Classic to use a font that can display your content.

Why Do I See Pipes and Squares in My Win32 AUT?

If you start up your application under test and see pipes and squares in the title bar, menus, or dialog boxes, it may mean that the operating system cannot support your application or that your system is not properly configured to display your content.

Why Do the Fonts on My System Look so Different?

Fonts that display in your menus, title bars and so on, are controlled by the registry settings and the **Display Properties > Appearance** settings of your computer.

If your fonts display too large or too small, you may have incorrectly set the appearance for an item:

1. Navigate to **Start > Settings > Control Panel > Display**.
2. Navigate to the **Appearance** tab and select **Windows** standard in the **Scheme** field.
3. Click **OK**.

Your desktop should now display normal.

Why Do Unicode Characters Not Display in the Silk Test Project Explorer

To view Unicode characters in the Silk Test Project Explorer, you must have installed a language pack with Unicode characters.

Why Is My Web Application Not Displaying Characters Properly?

If your Web application is not displaying the characters properly, or strange symbols or character are mixed in with your content, you may need to change a setting in your browser.

Internet Explorer Users

Check the settings for Encoding:

1. In Internet Explorer, click **View > Encoding**.
2. Select one of the following:
 - From the listed encodings, select one that meets the requirements of your application.
 - Click **More**, then select an encoding that meets the requirements of your application.
 - Click **Auto-Select**.

Mozilla Firefox Users

Check the settings for Character Coding:

1. In Mozilla Firefox, click **Settings > Content**.
2. In the **Fonts & Colors** section, click **Advanced**.
3. Select a character coding that meets the requirements of your application.

If you still have problems, ensure that your system locale is set for the language of your application under test.

File Formats

This section describes how you can troubleshoot issues with file formats in Unicode contents.

Why Am I Getting Compile Errors?

You may be trying to compile a file with an incompatible file format. Silk Test Classic supports three file formats: ANSI, UTF-8, and Unicode. If you try to compile files in Silk Test Classic that are in other formats, such as DBCS, you will get compile errors.

Workaround: In a Unicode-enabled text editor, save the file in one of the Silk Test Classic supported file formats: ANSI, UTF-8 or Unicode.

Why Does Silk Test Classic Open Up the Save As Dialog Box when I Try to Save an Existing File?

You have likely added content to the file that is incompatible with the file's existing file format. For example, you could have added Japanese characters to a frame file that was previously saved in ANSI format.

You must save the existing file in a compatible format.

Working with Input Method Editors

This section describes how you can troubleshoot issues when working with Input Method Editors (IMEs).

Why is English the Only Language Listed when I Click the Language Bar Icon?

You must be running an application, or area within the application, that supports an IME for a language other than English to be displayed in the Language bar icon. Applications that support IME include elements of Silk Test Classic such as include files and script files, Outlook, and Internet Explorer.

Why Does This IME Look so Different from Other IMEs I Have Used

IMEs can look different, depending on the operating system you are using and the particular IME you have accessed. For more information about IMEs, see Microsoft's support site.

Using Autocomplete

This section describes how you can automatically complete functions, members, application states, and data types.

Overview of AutoComplete

AutoComplete makes it easier to work with 4Test, significantly reducing scripting errors and decreasing the need to type text into your 4Test files by automatically completing functions, members, application states, and data types. There are four AutoComplete options:

Option	Description
Function Tip	Provides the function signature in a tooltip.
MemberList	Displays window children, properties, methods, and variables available to your 4Test file.
AppStateList	Displays a list of the currently defined application states.
DataTypeList	Displays a list of built-in and user-defined data types.

AutoComplete works with both Silk Test Classic-defined and user-defined 4Test files.

If you create a new 4Test file, you must name and save it as either a .t , .g.t, or .inc file in order for AutoComplete to work. After a 4Test file is saved, AutoComplete recognizes any changes you make to this file in the 4Test Editor and includes files that you reference through a 4Test use statement or the **Use Files** text box on the **Runtime Options** dialog box. When working with an existing 4Test file, you do not need to save or compile in order to access newly defined functions, methods, or members.

AutoComplete only works with 4Test files, which are .t, .g.t, and .inc files, that use hierarchical object recognition or dynamic object recognition with locator keywords.

AutoComplete does not work on comment lines or within plan, suite, or text files. AutoComplete does not support global variables of type window. However, AutoComplete supports Unicode content.

AutoComplete does not distinguish between Silk Test Classic Agents. As a result, AutoComplete displays all methods, properties, variables, and data types regardless of the Silk Test Classic Agent that you are using. For example, if you are using the Open Agent, functions and data types that work only with the Classic Agent are also displayed when you use AutoComplete. For details about which methods are supported for each Silk Test Classic Agent, review the corresponding .inc file, such as the `winclass.inc` file.

Customizing your MemberList

The members that you see in the MemberList depend on the MemberList options that you select. You can specify which members display in your MemberList. The members are window children, methods, properties, and variables. You can also determine how much detail is displayed in the MemberList by specifying the inheritance level and deciding whether you want to view class, data type, and function return type for methods in your MemberList.

All member options are enabled by default and the default inheritance level is below `AnyWin` class, meaning that methods for any class derived from the `AnyWin` class display in the MemberList. For additional information about the inheritance level, see the *General Options Dialog Box*.



Note: Methods that are defined in and above the `AnyWin` class, such as `Click` and `Exist`, which are defined in the `Winclass`, will not display in the MemberList. You can type these methods into your script, but they will not display in the MemberList unless you change the inheritance level to `All`.

To customize your MemberList:

1. Open Silk Test Classic and choose **Options > General**.
2. In the **AutoComplete** area of the **General Options** dialog box, make sure MemberList is selected.
3. In the **MemberList Options** area, select the members that you want to display in your MemberList. For example, if you want to view only properties and variables, uncheck the **Methods** and **Window Children** check boxes.
4. Select the appropriate Inheritance Level for the selected methods.

You can choose one of the following:

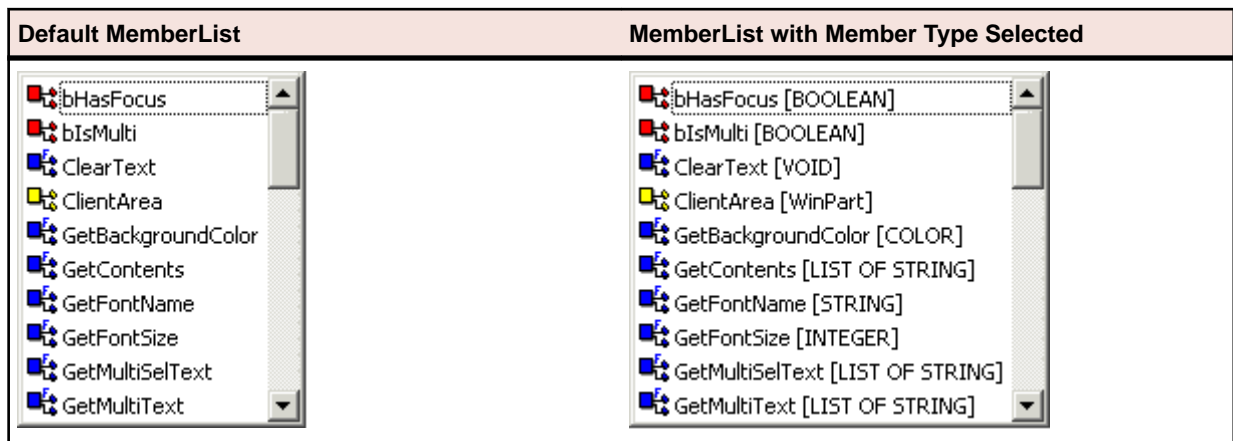
Below AnyWin Class Displays methods for any class derived from the `AnyWin` class. **Below AnyWin Class** is the default.

All Displays the complete inheritance for members all the way up through `AnyWin` and the control classes, including the `Winclass`.

None Displays only those members defined in the class of the current object and window declaration.

5. If you want to view attributes for the selected members, such as the class for window children, the data type for properties and variables, and the return type for method functions in your MemberList, check the **Member Type** check box.

Member Type is not checked by default. The following is a sample MemberList with and without **Member Type** checked.



6. Click **OK** on the **General Options** dialog box to save your changes.

Frequently Asked Questions about AutoComplete

Why isn't AutoComplete working?

AutoComplete only works with 4Test files with extension `.t`, `.g.t`, and `.inc`. If (untitled) is displayed in the title bar of your 4Test file, the file has not been saved yet. Save the file as `.t`, `.g.t`, or `.inc`.

After a 4Test file is saved, AutoComplete recognizes any changes you make to this file in the 4Test Editor and include files that you reference through a 4Test use statement or the **Use Files** text box on the **SilkTest Runtime Options** dialog box. Once you save a new file as a `.t`, `.g.t`, or `.inc`, you do not need to save or compile in order to access newly defined functions, methods, or members.

AutoComplete does not work on comment lines or within plan, suite, or text files.

Why doesn't a member display in my MemberList?

There are a few reasons you may not see a member in your MemberList. Here's what you should do:

1. On the **General Options** dialog box, make sure that you chose to show members of this type in the **MemberList Options** section. For additional information, see *Customizing your MemberList*.
2. Make sure the member you want to see is included in the inheritance level you selected. Below `AnyWin` class is the default; you might need to change your inheritance level to `All`. For additional information, see *Customizing your MemberList*.
3. Name and save your file with a `.t`, `.g.t`, or `.inc` extension.
4. Compile your file and fix any scripting errors. Anything following a compile error is not displayed in the MemberList or FunctionTip.


What happens if there is a syntax error in the current file?

Everything, based on the AutoComplete options you have selected, prior to the syntax error will display in your MemberList and/or FunctionTip. Anything following the syntax error will not display in your MemberList and/or FunctionTip. For additional information, see *Customizing your MemberList*.


What if I type something and AutoComplete does not find a match?

AutoComplete might not find a match for a number of reasons, for example because of the AutoComplete options you have specified or because of a compile error in your file. For information about fixing some of these issues, see *Customizing your MemberList* and *Turning AutoComplete Options Off*.

When AutoComplete does not find a match in the MemberList, focus remains on the first item in the list.

 **Note:** If you perform any of the selection methods, which means if you press Return, Tab, or click, the item will be pasted to the Editor.

You can simply type any function, method, or member in your 4Test files; AutoComplete does not restrict you in any way from typing in 4Test files.

 **Note:** You must dismiss the MemberList or FunctionTip before you can type in the Editor.

If you plan to use AutoComplete extensively, we recommend that you rename your identifiers in your window declarations. Knowing your identifier names helps, especially when working with long lists.

Why doesn't list of record type display in the FunctionTip?

This is a known limitation. FunctionTip does not support list of record types.

Why does AutoComplete show methods that are not valid for a 4Test class?

When using AutoComplete, the member list occasionally may reveal methods that are not valid for the 4Test class. The compiler will not catch these usage problems, but at Runtime the following exception is raised when the script is played back: `Error: Function <invalid method> is not defined for <window class>.`

Why does AutoComplete show methods, properties, variables, and data types that are not supported for the Silk Test Agent that I am using?

AutoComplete does not distinguish between Silk Test Agents. As a result, AutoComplete displays all methods, properties, variables, and data types regardless of the Silk Test Agent that you are using. For example, if you are using the Open Agent, functions and data types that work only with the Classic Agent are also displayed when you use AutoComplete. For detailed information about which methods are supported for each Silk Test Agent, review the corresponding .inc file, such as the `winclass.inc` file.

Turning AutoComplete Options Off

This topic contains instructions on how to disable AppStateList, DataTypeList, FunctionTip, and MemberList.

To turn off AutoComplete options:

1. Open Silk Test Classic and click **Options > General**.
2. In the **AutoComplete** area of the **General Options** dialog box, uncheck the check box for each of the AutoComplete options that you want to disable, and then click **OK**.

Using AppStateList

To display a list of currently defined application states:

1. Within your script, .t or .g.t, or within the include file, type your test case declaration, followed by the keyword `appstate` and then press **space**.

For example `testcase foo () appstate .`

A list of currently defined application states displays. You can also type the keyword `basedon` followed by a **space**. For example `appstate MyAppState () basedon .`

2. Use one of the following methods to select the appropriate member and paste it to the Editor.
 - Type the first letter or the first few letters of the member and then press **Enter** or **Tab**.
 - Use your arrow keys to locate the member and then press **Enter** or **Tab**.
 - Scroll through the list and click on a member to select it.

Using Data

To display a list of built-in and user-defined data types:

1. Within your script, .t or .g.t, or include file, type `array` or `varargs`, as appropriate, followed by the of keyword and a **space**.

For example, `list of`.

The current list of built-in and user-defined data types appears. You can also view the list of data types by pressing **F11**.

2. Use one of the following methods to select the appropriate member and paste it to the Editor:

- Type the first letter or the first few letters of the member and then press **Enter** or **Tab**.
- Use your arrow keys to locate the member and then press **Enter** or **Tab**.
- Scroll through the list and click on a member to select it.

Using FunctionTip

To display the function signature for a function, test case, or method.

1. Within your script, .t or .g.t, or include file, type the function, test case, or method name, followed by an open parenthesis " (".

For example `SetUpMachine(`. The function signature displays in a tooltip with the first argument, if any, in bold text. The function signature includes the return argument type, pass-mode, data type, name of the argument, and null and optional attributes, as they are defined.

2. Type the argument.

The FunctionTip containing the function signature remains on top and highlights the argument you are expected to enter in bold text. As you enter each argument and then type a comma, the next argument that you are expected to type is highlighted. The expected argument is always indicated with bold text; if you backspace or delete an argument within your function, the expected argument is updated accordingly in the FunctionTip. The FunctionTip disappears when you type the close parenthesis ") " to complete the function call.

If you want to dismiss the FunctionTip, press **Escape**. FunctionTip is enabled by default. See *Turning AutoComplete Options Off* if you want to disable FunctionTip.

Using MemberList

This topic contains instructions on how to use MemberList to view and select a list of members.

To view a list of members:

1. Customize the member list so that it displays the information you require.

You can choose to display any or all of the following members:

Member	Description
Window children	Displays all window objects of type WINDOW that are defined in window declarations in the referenced .t, .g.t, and .inc files. Indicated in the MemberList with a yellow icon.
Methods	Displays all methods defined in the referenced .t, .g.t, and .inc files. Indicated in the MemberList with a blue icon.
Properties	Displays all properties defined in the referenced .t, .g.t, and .inc files. Indicated in the MemberList with a red icon.

Member	Description
Variables	Displays all defined variables in the referenced .t, .g.t, and .inc files, including native data types, data, and records. Fields defined for records and nested records also display in the list. Indicated in the MemberList with a red icon.

2. Within your script or include file, type the member name and then type a period (.).

For example `Find.`

The MemberList displays. Depending on the MemberList Options and the Inheritance Level you select, the types of members that display in the MemberList will vary.

3. Use one of the following methods to select the appropriate member and paste it to the Editor:

- Type the first letter or the first few letters of the member and then press **Enter** or **Tab**.
- Use your arrow keys to locate the member and then press **Enter** or **Tab**.
- Scroll through the list and click on a member to select it.

The MemberList is case sensitive. If you type the correct case of the member, it is automatically highlighted in the MemberList; press **Enter** or **Tab** once to paste it to the Editor. If you do not type the correct case, the member has focus, but is not highlighted; press **Enter** or **Tab** twice to select the member and paste it to the Editor. To dismiss the MemberList, press **Escape**.

Overview of the Library Browser

Click **Help** > **Library Browser** to access the **Library Browser**. It provides online documentation for:

- Built-in 4Test methods, properties, and functions: the **Library Browser** shows the name and class of the method, one line of descriptive text, syntax, and a list of parameters, including a description.
- User-defined methods: the **Library Browser** shows the name and class of the method, syntax, and a list of parameters. It displays User defined as the method description and displays the data type for each parameter.
- User-defined Properties: As with user-defined methods, the description for user-defined properties by default is User defined.

The **Library Browser** does not, by default, provide documentation for your user-defined functions. You can add to the contents of the **Library Browser** to provide descriptive text for your user-defined methods, properties, and functions.

Library Browser Source File

The core contents of the **Library Browser** are based on a standard Silk Test Classic text file, `4test.txt`, which contains information for the built-in methods, properties, and functions.

You can edit `4test.txt` to include your user-defined information, or define your site-specific information in one or more separate files, and then have Silk Test Classic compile the file (creating `4test.hlp`) to make it available to the **Library Browser**. Information about methods in `4test.hlp` is also used in the **Verify Window** dialog box for methods.

Silk Test Classic does not update `4test.txt` with user-defined information; instead it populates the **Library Browser** from information it receives when include files are compiled in memory. You modify `4test.txt` to override the default information displayed for user-defined objects.

Simply looking through `4test.txt` should give you all the help you need about how to structure the information in the file. The following table lists all the keywords and describes how they are used in `4test.txt`. You should edit a copy of `4test.txt` to add the information you want.

Keywords

Keywords are followed by a colon and one or more spaces.

class	Name of the class.
function	Name of the function. Specify the full syntax. If the function returns a value, specify: <code>return_value = function_name (parameters)</code> Otherwise, specify: <code>function_name (parameters)</code>
group	Name of the function category.
method	Description of the method. Specify the full syntax. If the method returns a value, specify: <code>return_value = method_name (parameters)</code> Otherwise, specify: <code>method_name (parameters)</code>
notes	Description of the method, property, or function, up to 240 characters. Do not split the description into multiple notes fields, since only the first one is displayed.
parameter	Name and description of a method or function parameter. Each parameter is listed on its own line. Specify the name, followed by a colon, followed by the description of the parameter.
property	Name of the property.
returns	Type and description of the return value of the method or function. Specify the name, followed by a colon, followed by the description of the return value.
#	Comment.

Adding Information to the Library Browser

1. Make a backup copy of the default `4test.txt` file, which is in the directory where you installed Silk Test Classic, and store your backup copy in a different directory.
2. In an ASCII text editor, open `4test.txt` in your Silk Test Classic installation directory and edit the file. See examples for methods, properties, and functions, if necessary.
3. Quit Silk Test Classic.
4. Place your modified `4test.txt` file in the Silk Test Classic installation directory.
5. Restart Silk Test Classic. Silk Test Classic sees that your source file is more recent than `4test.hlp` and automatically compiles `4test.txt`, creating an updated `4test.hlp`. If there are errors, Silk Test Classic opens a window listing them and continues to use the previous `4test.hlp` file for the Library Browser. If there were errors, fix them in `4test.txt` and restart Silk Test Classic. Your new definitions are displayed in the **Library Browser** (assuming that the files containing the declarations for your custom classes, methods, properties, and functions are loaded in memory).

There is another approach to updating the **Library Browser**: maintain information in different source files.

If the **Library Browser** isn't displaying your user-defined objects, close the **Library Browser**, recompile the include files that contain your user-defined objects, then reopen the **Library Browser**.

Add User-Defined Files to the Library Browser with Silk Test Classic

1. Create a text file that includes information for all your custom classes and functions, using the formats described in the **Library Browser** source file. If you have added methods or properties to built-in classes, you should add that information in the appropriate places in `4test.txt`, as described above. Only document your custom classes and functions in your own help file.
2. Click **Options > General** and add your help file to the list in the **Help Files For Library Browser** field. Separate the files in this list with commas.
3. Click **OK**. Silk Test Classic recompiles `4test.hlp` to include the information in all the files listed in the **Help Files For Library Browser** field. If there are errors, Silk Test Classic opens a window listing them and continues to use the previous `4test.hlp` file for the **Library Browser**. If you had errors, fix them in your source file, then quit and restart Silk Test Classic. Silk Test Classic recompiles `4test.hlp` using your modified source file.

Viewing Functions in the Library Browser

To view information about built-in 4Test functions in the **Library Browser**:

1. Click **Help > Library Browser**, and then click the **Functions** tab.
2. Select the category of functions you want in the **Groups** list box. To see all built-in 4Test functions, check the **Include all** check box.
Functions are listed in the **Functions** list box.
3. Select the function for which you want information.

Viewing Methods for a Class in the Library Browser

4Test classes have methods and properties. When you select the **Methods** or **Properties** tabs in the **Library Browser**, you see a list of all the built-in and user-defined classes in hierarchical form.

To see the methods or properties for a class:

1. Click **Help > Library Browser**, and then click the **Methods** or **Properties** tab.
2. Select the class in the **Classes** list box.
Double-click a + box to expand the hierarchy. Double-click a – box to collapse the hierarchy. The methods or properties for the selected class are displayed. By default, only those methods or properties that are defined by the class are displayed. To see all methods or properties that are available to the class (that is, methods or properties also defined by an ancestor of the class), select the **Include inherited** check box. To see all methods or properties (even those not available to the selected class), select the **Include all** check box.
3. Select a method or property. Information about the selected method or property is displayed.

If the **Library Browser** is not displaying your user-defined objects, close the **Library Browser**, recompile the include files that contain your user-defined objects (**Run > Compile**), and then re-open the **Library Browser**.

Examples of Documenting User-Defined Methods

This topic contains examples of adding user-defined methods, properties, and functions to the **Library Browser**.

```
#####  
class:      DialogBox
```

```

...

## custom method
method: VerifyNumChild (iExpectedNum)
parameter: iExpectedNum: The expected number of child objects (INTEGER).
notes: Verifies the number of child objects in a dialog box.

Documenting user-defined properties: Add the property descriptions to the
appropriate class section in 4test.txt, such as:

##*****
class: DialogBox
...

## custom property
property: iNumChild
notes: The number of child objects in the dialog box.

Documenting user-defined functions: Create a group called User-defined
functions and document your functions, such as:
group: User-defined functions

function: FileOpen (sFileName)
parameter: sFileName = "myFile": The name of the file to open.
notes: Opens a file from the application.

function: FileSave (sFileName)
parameter: sFileName = "myFile": The name of the file to save.
notes: Saves a file from the application.


```

Web Classes Not Displayed in Library Browser

This functionality is supported only if you are using the Classic Agent.

Problem

The class hierarchy in the **Library Browser** does not include the Web classes, which are `BrowserChild`, `HtmlText`, and so on.

Possible Causes and Solutions

- No browser extension is enabled.** Make sure that at least one browser extension is enabled.
- Enhanced support for Visual Basic is enabled.** Disable Visual Basic by un-checking the **ActiveX** check box for the Visual Basic application in the **Extension Enabler** and **Extensions** dialog boxes.

Library Browser

Use **Library Browser** to quickly look up documentation for built-in and user-defined 4Test methods, properties, and functions.

Click **Help > Library Browser**.

Methods tab

Click the **Methods** tab to display information about the built in and user-defined 4test methods.

- Classes list** Click the class to display. A list of methods for the class is displayed.
- Include all** Check to view all the 4Test methods available for the selected class.

Methods list Click the method to display. Detailed information is displayed about the selected method and class.

Include inherited Check to view the inherited 4Test methods available for the selected class.

Properties tab

Click the **Properties** tab to display information about the built-in and user-defined properties.

Classes list Click the class to display. A list of properties for the class is displayed.

Include all Check to view all of the 4Test properties available.

Properties list Click the property to display. Detailed information is displayed about the selected property and class.

Include inherited Check to view the inherited 4Test properties available for the selected class.

Functions tab

Click the **Functions** tab to display information about the built-in and user-defined functions.

Groups list Click the group to display. A list of functions for the selected group is displayed.

Include all Check to view all of the 4Test methods available.

Functions list Click the function to display. Detailed information is displayed about the selected group and function.

Text Recognition Support

Text recognition methods enable you to conveniently interact with test applications that contain highly customized controls, which cannot be identified using object recognition. You can use *text clicks* instead of coordinate-based clicks to click on a specified text string within a control.

For example, you can simulate selecting the first cell in the second row of the following table:

CustomerName	FirstOrder	ID	IsActive	CreditCard
Bob Villa	01.01.2008	0	<input checked="" type="checkbox"/>	MasterCard
Brian Miller	02.01.2008	1	<input type="checkbox"/>	Visa
Caral Rudd	03.01.2008	2	<input checked="" type="checkbox"/>	American Ex...
Dan Rundgren	04.01.2008	3	<input type="checkbox"/>	MasterCard
Devie Yingstein	05.01.2008	4	<input checked="" type="checkbox"/>	Visa

Specifying the text of the cell results in the following code:

```
table.TextClick("Brian Miller")
```

Text recognition methods are supported for the following technology domains:

- Win32.
- WPF.
- Windows Forms.
- Java SWT and Eclipse.
- Java AWT/Swing.



Note: For Java Applets, and for Swing applications with Java versions prior to version 1.6.10, text recognition is supported out-of-the-box. For Swing applications with Java version 1.6.10 or later,

which do not support Direct3D, you have to add the following command-line element when starting the application:

```
-Dsun.java2d.d3d=false
```

For example:

```
javaw.exe -Dsun.java2d.d3d=false -jar mySwingApplication.jar
```

Text recognition is not supported for Java Applets and Swing applications that support Direct3D.

- Internet Explorer.



Note: Text recognition does not work with controls that are not visible on the screen. For example, you cannot use text recognition for a text that is scrolled out of view.



Note: Text recognition might not work if the font that is used in the target text is not installed on the machine on which the test is executed.

Text recognition methods

The following methods enable you to interact with the text of a control:

- TextCapture** Returns the text that is within a control. Also returns text from child controls.
- TextClick** Clicks on a specified text within a control. Waits until the text is found or the *Object resolve timeout*, which you can define in the synchronization options, is over.
- TextRectangle** Returns the rectangle of a certain text within a control or a region of a control.
- TextExists** Determines whether a given text exists within a control or a region of a control.

Text click recording

Text click recording is enabled by default. To disable text click recording, click **Options > Recorder > Recording** and uncheck the **OPT_RECORD_TEXT_CLICK** check box.

When text click recording is enabled, Silk Test Classic records `TextClick` methods instead of clicks with relative coordinates. Use this approach for controls where `TextClick` recording produces better results than normal coordinate-based clicks. You can insert text clicks in your script for any control, even if the text clicks are not recorded.

If you do not wish to record a `TextClick` action, you can turn off text click recording and record normal clicks.

The text recognition methods prefer whole word matches over partially matched words. Silk Test Classic recognizes occurrences of whole words previously than partially matched words, even if the partially matched words are displayed before the whole word matches on the screen. If there is no whole word found, the partly matched words will be used in the order in which they are displayed on the screen.

Example

The user interface displays the text *the hostname is the name of the host*. The following code clicks on *host* instead of *hostname*, although *hostname* is displayed before *host* on the screen:

```
control.TextClick("host")
```

The following code clicks on the substring *host* in the word *hostname* by specifying the second occurrence:

```
control.TextClick("host", 2)
```


Slowing Down Tests

This functionality is supported only if you are using the Open Agent.

Some applications under test might require extensive loading of application data in the UI, and might not be finished on time with loading objects that are required for replaying a test. To successfully replay tests on such an AUT, you can check for the existence of an object before performing an action on it, or you can add sleeps before performing an action.



Note: Micro Focus does not recommend generally adding sleeps to tests, because in most cases Silk Test Classic will automatically detect if an object is available, and sleeps might severely reduce the performance of tests.

1. To check if an object is available in the AUT, use the Exists method.

For example, to wait for six seconds for the button INPUT to become available, add the following line to your test script:

```
WebBrowser.BrowserWindow.Input.Exists(6)
```

2. To add a sleep before performing an action on a control, use the method.

For example, to sleep for six seconds, add the following line to your test script:

```
Sleep(6)
```

Testing Applications in Multiple UI Sessions on a Single Machine

This functionality is supported only if you are using the Open Agent.

To test applications in multiple UI sessions on a single machine or to test multiple agents on a single machine, connect to multiple Open Agent instances on the machine. Every agent runs in its own UI-session. A UI session can be a Remote Desktop Protocol (RDP) connection or a Citrix-based connection.

1. Create the UI sessions.
2. Open a command line window.
3. Navigate to the folder `/ng/MultiSessionLauncher` in the Silk Test installation directory.

For example, the default folder path might look like the following: `C:\Program Files (x86)\Silk\SilkTest\ng\MultiSessionLauncher`.

4. In each UI session, execute the following command:

```
MicroFocus.SilkTest.MultiSessionLauncher.exe <port>.
```



Note: Use a unique port number, because this port will be used in your Silk Test Classic script to identify the Open Agent and the UI session in which the agent is running.


5. Change your Silk Test Classic scripts to connect to the Open Agent instances.


To connect to an Open Agent instance, add the following line to the script:

```
HMACHINE hAgent = Connect("hostname:port")
```

Where *hostname* is the name of the machine on which the agent is running, and *port* is the unique port that you have used to execute the launcher.

The resulting objects are independent of each other and can be used either in one thread or in multiple threads.

 **Note:** If you want to launch an application in multiple UI sessions, you have to execute the base state for each UI session.

 **Note:** To use TrueLog when testing applications in multiple UI sessions on a remote machine, you need to manually copy any generated TrueLog files from the remote machine to your local machine.

Example

Assume that the server machine that is hosting the UI sessions is named *ui-srv*. You can create three UI sessions by using the ports 22902, 22903, and 22904.

In the first session, open the command line window, navigate to the `MultiSessionLauncher` directory, and type the following:

```
MicroFocus.SilkTest.MultiSessionLauncher.exe 22902
```

Do the same for the other two sessions with the respective ports 22903 and 22904.

To connect to the Open Agent instances, add the following code to your script:

```
HMACHINE hAgent1 = Connect("ui-srv:22902")
HMACHINE hAgent2 = Connect("ui-srv:22903")
HMACHINE hAgent3 = Connect("ui-srv:22904")
```

The following sample script prints a simple text to each of the three UI sessions:

```
[ - ] main()
    [ ] HMACHINE hAgent1 = Connect("ui-srv:22902")
    [ ] HMACHINE hAgent2 = Connect("ui-srv:22903")
    [ ] HMACHINE hAgent3 = Connect("ui-srv:22904")

    [ ] hAgent1->FindWindow("//Window").TypeKeys("Hello to
session 1!")
    [ ] hAgent2->FindWindow("//Window").TypeKeys("Hello to
session 2!")
    [ ] hAgent3->FindWindow("//Window").TypeKeys("Hello to
session 3!")
```

Running Tests and Interpreting Results

This section describes how you can run your tests and interpret the generated results.

Running Tests

This section describes how you can run your tests with Silk Test Classic.

Creating a Suite

After you have created a number of script files, you might want to collect them into a test suite. A suite is a file that names any number of scripts. Instead of running each script individually, you run the suite, which executes in turn each of your scripts and all the test cases they contain. Suite files have a `.s` extension.

1. Click **File > New**.
2. Select the **Suite** radio button and click **OK**. An untitled suite file is displayed.
3. Enter the names of the script files in the order you want them executed. For example, the following suite file executes the `find.t` script first, the `goto.t` script second, and the `open.t` script third:

```
find.t  
goto.t  
open.t
```

4. Click **File > Save** to save the file.
5. If you are working within a project, you are prompted to add the file to the project. Click **Yes** if you want to add the file to the open project, or **No** if you do not want to add this file to the project.

Passing Arguments To a Script

You can pass arguments to a script. For example, you might want to pass in the number of iterations to perform or the name of a data file. All functions and test cases in the script have access to the arguments.

How to pass arguments to a script

All arguments are passed in as strings, separated by spaces, such as: Bob Emily Craig

If an argument is more than one word, enclose it with quotation marks. For example, the following passes in three arguments: "Bob H" "Emily M" "Craig J"

You can pass arguments to a script using the following methods:

- Specify the arguments in the **Arguments** field in the **Runtime Options** dialog box. To open the dialog box, click **Options > Runtime** in the menu bar.
- The **Arguments** field in the **Run Test Case** dialog box is used to pass arguments to a test case, not to an entire script.
- Specify the arguments in a suite file after a script name, for example `find.t arg1 arg2`.
- Provide arguments when you invoke Silk Test Classic from the command line.
- If you pass arguments in the command line, the arguments provided in the command line are used and any arguments specified in the currently loaded options set are not used. To use the arguments in the currently loaded options set, do not specify arguments in the command line.

Processing arguments passed into a test script

You use the `GetArgs` function to process arguments passed into a script. `GetArgs` returns a list of strings with each string being one of the passed arguments. Any test case or function in a script can call `GetArgs` to access the arguments.

Example: Passed arguments

The following test case prints a list of all the passed arguments:

```
testcase ProcessArgs ( )
LIST OF STRING lsArgs
lsArgs = GetArgs ( )
ListPrint (lsArgs)

//You can also process the arguments individually. The following test case
prints the second argument passed:
testcase ProcessSecondArg ( )
LIST OF STRING lsArgs
lsArgs = GetArgs ( )
Print (lsArgs[2])

//The following testcase adds the first two arguments:
testcase AddArgs ( )
LIST OF STRING lsArgs
lsArgs = GetArgs ( )
NUMBER nArgSum

nArgSum = Val (lsArgs[1]) + Val (lsArgs[2])
Print (nArgSum)
```

You can use the `Val` function to convert the arguments, which are always passed as strings, into numbers.

When the arguments script 10 20 30 are passed to the `scr_args.t` script, the test result is:

```
Script scr_args.t (10, 20, 30) - Passed
Passed: 1 test (100%)
Failed: 0 tests (0%)
Totals: 1 test, 0 errors, 0 warnings

Testcase AddArgs - Passed

30
```

Running a Test Case

When you run a test case, Silk Test Classic interacts with the application by executing all the actions you specified in the test case and testing whether all the features of the application performed as expected.

Silk Test Classic always saves the suite, script, or test plan before running it if you made any changes to it since the last time you saved it. By default, Silk Test Classic also saves all other open modified files whenever you run a script, suite, or test plan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog box.

1. Make sure that the test case that you want to run is in the active window.
2. Click **Run Testcase** on the **Basic Workflow** bar.
If the workflow bar is not visible, choose **Workflows > Basic** to enable it.
Silk Test Classic displays the **Run Testcase** dialog box, which lists all the test cases contained in the current script.
3. Select a test case and specify arguments, if necessary, in the **Arguments** field.
Remember to separate multiple arguments with commas.

4. To wait one second after each interaction with the application under test is executed, check the **Animated Run Mode (Slow-Motion)** check box.

Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed:

- BaseStateExecutionFinished
- Connecting
- Verify
- Exists
- Is
- Get
- Set
- Print
- ForceActiveXEnum
- Wait
- Sleep

5. To view results using the TrueLog Explorer, check the **Enable TrueLog** check box. Click **TrueLog Options** to set the options you want to record.

6. Click **Run**. Silk Test Classic runs the test case and generates a results file.

For the Classic Agent, multiple tags are supported. If you are running test cases using other agents, you can run scripts that use declarations with multiple tags. To do this, check the **Disable Multiple Tag Feature** check box in the **Agent Options** dialog box on the **Compatibility** tab. When you turn off multiple-tag support, 4Test discards all segments of a multiple tag except the first one.

7. *Optional:* If necessary, you can press both **Shift** keys at the same time to stop the execution of the test.

Running a Test Plan

Before running a test plan, make sure that the window declarations file for the test plan is correctly specified in the **Runtime Options** dialog box and that the test plan is in the active window.

- To run the entire test plan, click **Run > Run All Tests**. Silk Test Classic runs each test case in the plan and generates a results file.
- To run only tests that are marked, click **Run > Run Marked Tests**. Silk Test Classic runs each marked test and generates a results file.

You can also run a single test case without marking it.

If your test plan is structured as a master plan and associated sub-plans, Silk Test Classic automatically opens any closed sub-plans before running. Silk Test Classic always saves the suite, script, or test plan before running it if you made any changes to it since the last time you saved it. By default, Silk Test Classic also saves all other open modified files whenever you run a script, suite, or test plan. To prevent this automatic saving of other open modified files, uncheck the **Save Files Before Running** check box in the **General Options** dialog box.

To stop the execution of a test plan, press both **Shift** keys at the same time.

Running the Currently Active Script or Suite

1. Make sure the script or suite you want to run is in the active window.
2. Choose **Run > Run**.


Silk Test Classic runs all the test cases in the script or suite and generates a results file.


Running Tests in Parallel

This functionality is supported only if you are using the Open Agent.

You can use the command line to execute tests against mobile applications in parallel against multiple devices. This functionality is supported for both mobile web and native mobile applications, and can be used to run tests on physical devices, Android emulators, and iOS simulators. For example, you can use this functionality when executing test from a continuous integration server, or from Silk Central.

To enable parallel test replay, set the environment variable `SILKTEST_ENABLE_PARALLEL_TESTING` to true.

 **Note:** Enabling parallel testing causes the Open Agent to handle each test-executing process separately. Applications which have been tested in one Silk Test client cannot be tested from another client, while the initial client is running. For example, you cannot test the same application alternating between Silk4J and JUnit.


 **Note:** You cannot execute multiple test runs on the same mobile device at the same time. Before running tests in parallel, ensure that enough devices or emulators are available. Any test runs that get no mobile device or emulator assigned will fail.

Each parallel test run starts as a separate Partner process. Multiple processes starting simultaneously might each try to start the Open Agent on the machine on which Silk Test Classic is running. Running the Open Agent multiple times on the same machine is not possible and will cause Silk Test Classic to throw an exception. To avoid this, ensure that the Open Agent is running before starting the parallel test runs.

For example, you could start the Open Agent on the local machine by using the following PowerShell command:

```
powershell.exe -noprofile -command  
\"[System.Reflection.Assembly]::LoadWithPartialName('SilkTest.Ntf');  
[SilkTest.Ntf.Agent]::Connect('localhost')
```

The test results are stored in multiple TrueLog files, one for each test run. To ensure that the TrueLog files are not overwritten, you can add placeholders to the TrueLog file name. For additional information, see [Setting TrueLog Options](#).

 **Note:** If you are experiencing high memory consumption during testing, ensure that test results are saved in the compressed TLZ file format, and not in the XLG format.

Setting Replay Options for the Open Agent

This functionality is supported only if you are using the Open Agent.

You can set the replay options to optimize replaying tests with the Open Agent.

1. Click **Options > Agent**. The **Agent Options** dialog box appears.
2. Click the **Replay** tab.
3. Select a **Replay mode** from the list.
 - **Default:** Use this mode for the most reliable results. Each control uses the best method for the control type, by using either the mouse and keyboard (low level) mode or the API (high level) mode.
 - **High level:** Use this mode to replay each control using the API.
 - **Low level:** Use this mode to replay each control using the mouse and keyboard.
4. To ensure that the window is active before a call is executed, check the **Ensure window is active** check box.
5. Click **OK**.

Stopping a Running Test Case Before it Completes

To stop running a test case before it completes:

- If your test application is on a target machine other than the host machine, click **Run > Abort**.
- If your test application is running on your host machine, press `Shift+Shift`.

Setting a Test Case to Use Animation Mode

To slow down a test case during playback so that it can be observed, set the test case to use *animation mode*. For instance, if you want to demonstrate a test case to someone else, you might want to use animation mode.

You can specify the animation mode when you run a test case, or you can specify the animation mode in the **Runtime Options** dialog.

To specify the animation mode using the **Runtime Options** dialog:

1. From the main menu, click **Options > Runtime**.
2. In the **Runtime Options** dialog, check the **Animated Run Mode (Slow-Motion)** check box.
3. Click **OK**.

Run Application State Dialog Box

Use the **Run Application State** dialog box to run or debug an application state defined in your test frame file or active script file.

Click **Run > Application State**.

Application state	Displays the application states defined in the active frame or script file.
Run	Runs the selected application state.
Debug	Opens the file in which the application state resides and enters debugging mode. Debugging mode makes all the debugging commands available on the Breakpoint , Debug , and View menus.
Cancel	Closes the dialog box.

Run Testcase Dialog Box

Use the **Run Testcase** dialog box to run or debug a test case accessible from the active script file. This dialog box is only available when the active window contains either a test case or a results file in which the current line is a test case or a data-driven test case. Data-driven test cases begin with "DD_". You cannot run a test case with both arguments and rows.

Click **Run > Testcase** or **Testplan > Detail > Specify Rows**.

Testcase	Displays all the test cases accessible from the current script.
Arguments	Allows you to enter arguments to pass to the selected test case.
Animated Run Mode (Slow-Motion)	Check to wait one second after each interaction with the application under test is executed. Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. Executions of the default base state and functions that include one of the following strings are not delayed: <ul style="list-style-type: none">• <code>BaseStateExecutionFinished</code>

- Connecting
- Verify
- Exists
- Is
- Get
- Set
- Print
- ForceActiveXEnum
- Wait
- Sleep

Use Sample Data from Script	Check to run the test case once using the sample record for each table used by the test case. This is available only if the selected test case is a data-driven test case.
Enable TrueLog	To view results using the TrueLog Explorer, check the Enable TrueLog check box. This check box enables you to enable or disable TrueLog Explorer each time you run a test case. You can also check this check box in the TrueLog Options dialog box. When you enable or disable TrueLog Explorer in the Run Test Case dialog box, the same changes are made in the TrueLog Options dialog box. Likewise, when you enable or disable TrueLog Explorer in the TrueLog Options dialog box, the same changes are made in the Run Testcase dialog box.
TrueLog Options	Click to set the TrueLog options you want to record. By default, Silk Test Classic uses the options that were most recently specified in the TrueLog Options dialog box. Check the Enable TrueLog check box to enable this button.
Run	Runs the selected test case. For data-driven test cases, the argument must be in the test case record format. If the Arguments text box is empty, Run causes the test case to run all combinations of all rows in all tables used by the test case.
Debug	Opens the file in which the test case resides, enters debugging mode, and places a breakpoint at the first line of the test case. Debugging mode makes available all the debugging commands on the Breakpoint , Debug and View menus. If the Arguments list is empty, Debug runs the data-driven test case using all rows, but in debug mode. Since the default breakpoint is the <code>testcase</code> statement, by default the debugger skips <code>DefaultRunDataDrivenTestCase()</code> and stops at the test case with the first set of rows as input. The debugger acts as described for Debug in Run > Testcase , but with data-driven test cases only sets of selected rows are passed to the test case.
Cancel	Closes this dialog box.
Specify Rows	Opens the Specify Rows dialog box where you can select the rows you want to run from the tables for this data-driven test case. This is available only if you have selected Use Sample Data from Script for a data-driven test case.

Runtime Status Dialog Box

By default, this dialog box automatically appears on the partner machine while executing a test case, script or test plan. To keep this status dialog box automatically hidden at runtime, click **Options > Runtime** and in the **Execution** area clear the **Show Detailed Status Window** check box. This dialog box is for status purposes only.

If you are running a script, test case or test plan on a target machine other than the host machine, you can safely close the **Runtime Status** dialog box once execution begins. To reopen this dialog box after closing it, or if it is hidden by default, click **Run > Show Status**. This menu option is available only if running a script, test case, or test plan.

This dialog box is display only.

Program	Lists the recent tasks the agent executed as defined in the script. As Silk Test Classic executes the plan the list will change. Only the first item listed will remain constant. That is, if a test plan is executed the name of that file will always appear at the top of the Program list. If you are running a test plan that is a masterplan, which means it includes other test plans, the masterplan name will always be the first item listed, not the sub-plans.
Elapsed	Lists how long it took to execute the corresponding task listed under Program . The first item in the list is the current cumulative time elapsed for the entire execution.
Errors	Lists how many errors occurred while executing the corresponding task listed in Program . The first item in the list is the current cumulative number of errors for the entire execution.
Current agent call	Displays the current function the agent is executing against the AUT.
Last error	Displays the last error encountered.

Interpreting Results

This section describes how you can use the Difference Viewer, the results file, and the reports to interpret the results of your tests.

Overview of the Results File

A results file provides information about the execution of the test case, script, suite, test plan, or keyword-driven test. By default, the results file has the same name as the executed script, suite, test plan, or keyword-driven test, but with a `.res` extension. For example, `find.res`.

Whenever you run tests, Silk Test Classic generates a results file, which indicates how many tests passed and how many failed, describes why tests failed, and provides summary information. You can invoke comparison tools from within the results file that pinpoint exactly how the runtime results differ from your known baselines. Test-plan results files offer additional features, such as the ability to generate a Pass/Fail report or compare different runs of the test plan. When Silk Test Classic displays a results file, on the menu bar it includes the **Results** menu, which allows you to manipulate the results file and locate errors. The **Results** menu appears only when the active window displays a results file.

TrueLog Explorer

Silk Test Classic also provides the TrueLog Explorer to help you analyze test results files. You must configure Silk Test Classic to use the TrueLog Explorer and specify what you want to capture.

Multiple User Environments

A `.res` file can be opened by multiple users, as long as no test is in process. This means you cannot have two users run tests at the same time and write to the same results file. You can run a test on the machine while the file is open on the other machine. However, you must not add comments to the file on the other machine, or you will corrupt the `.res` file and will not be able to report the results of the test. If you add comments to the file on both machines, the comments will be saved only for the file that is closed (and therefore saved) first.

Default Settings

By default, the results file displays an overall summary at the top of the file, including the following:

- The name of the script, suite, test plan, or keyword-driven test.

- The machine on which the tests were executed.
- The amount of tests that were executed.
- The amount of errors and warnings that were generated during the execution.
- The actual errors that were generated.
- Timing information.

To hide the overall summary, click the summary and click **Results > Hide Summary**. For a script or suite results file, the individual test summaries contain timing information and errors or warnings. For a test plan results file, the individual test summaries contain the same information as in the overall summary plus the name of the test case and script file.

While Silk Test Classic displays the most current version of the script, suite, or test plan, by default Silk Test Classic saves the last five sets of results for each script, suite, or test plan executed. To change the default number, use the **Runtime Options** dialog. As results files grow after repeated testing, a lot of unused space can accumulate in the files. You can reduce the size of a results file with the Compact menu option.

The format for the rest of a test plan results file follows the hierarchy of test descriptions that were present in the test plan. Test statements in the test plan that are preceded by a pound sign (#) as well as comments, which are using the `comment` statement, are also printed in the results file, in context with the test descriptions.

```
comment
```

To change the default name and directory of the results file, edit the **Runtime Options** dialog.



Note: If you provide a local or remote path when you specify the name of a results file in the **Directory/Field** field on the **Runtime Options** dialog, the path cannot be validated until script execution time.

Viewing Test Results

Whenever you run tests, a results file is generated which indicates how many tests passed and how many failed, describes why tests failed, and provides summary information.

1. Click **Explore Results** on the **Basic Workflow** or the **Data Driven Workflow** bars.
2. On the **Results Files** dialog box, navigate to the file name that you want to review and click **Open**.

By default, the results file has the same name as the executed script, suite, or test plan. To review a file in the TrueLog Explorer, open a `.xlg` file. To review a results file, open a `.res` file.

Errors And the Results File

You can expand the text of an error message or have Silk Test Classic find the error messages for you. To navigate from a test plan test description in a results file to the actual test in the test plan, click the test description and select **Results > Goto Source**.

Navigating to errors in the script

There are several ways to move from the results file to the actual error in the script:

- Double-click in the margin next to an error line to go to the script file that contains the 4Test statement that failed.
- Click an error message and select **Results > Goto Source**.
- Click an error message and press **Enter**.

What the box icon means

Some expanded error messages are preceded by a box icon and three asterisks.

If the error message relates to an application's behavior, as in `Verify selected text failed`, Silk Test Classic opens the **Difference Viewer**. The **Difference Viewer** compares actual and expected values for a given test case.

Application appearance errors

When you click a box icon followed by a bitmap-related error message, the bitmap tool starts, reads in the baseline and result bitmaps, and opens a **Differences** window and **Zoom** window.

Bitmap tool

In the **Bitmap Tool**:

- The baseline bitmap is the bitmap that is expected, which means the baseline for comparison.
- The results bitmap is the actual bitmap that is captured.
- The **Differences** window shows the differences between the baseline and result bitmap.

The **Bitmap Tool** supports several comparison commands, which let you closely inspect the differences between the baseline and results bitmaps.

Finding application logic errors

To evaluate application logic errors, use the **Difference Viewer**, which you can open by clicking the box icon following an error message relating to an application's behavior.

The Difference viewer

Clicking the box icon opens the **Difference Viewer**'s double-pane display-only window. It lists every expected (baseline) value in the left pane and the corresponding actual value in the right pane.

All occurrences are highlighted where expected and actual values differ. On color monitors, differences are marked with red, blue, or green lines, which denote different types of differences, for example, deleted, changed, and added items.

When you have more than one screen of values or are using a black-and-white monitor, use **Results > Next Result Difference** to find the next difference. Use **Update Expected Values**, described next, to resolve the differences.

Updating expected values

You might notice upon inspecting the **Difference Viewer** or an error message in a results file that the expected values are not correct. For example, when the caption of a dialog changes and you forget to update a script that verifies that caption, errors are logged when you run the test case. To have your test case run cleanly the next time, you can modify the expected values with the **Update Expected Value** command.



Note: The **Update Expected Value** command updates data within a test case, not data passed in from the test plan.

Debugging tools

You might need to use the debugger to explore and fix errors in your script. In the debugger, you can use the special commands available on the **Breakpoint**, **Debug**, and **View** menus.

Marking failed test cases

When a test plan results file shows test case failures, you might choose to fix and then rerun them one at a time. You might also choose to rerun the failed test cases at a slower pace, without debugging them, simply to watch their execution more carefully.

To identify the failed test cases, make the results file active and select **Results > Mark Failures in Plan**. All failed test cases are marked and test plan file is made the active file.

Viewing Differences

If a verification fails, because the actual value is different to the expected value, you can use the **Difference Viewer** from the results file to compare the expected value with the actual value.

You can use **Results > Next Result Difference** to find the next difference and update the values using **Results > Update Expected Value**.



Note: The **Difference Viewer** does not work for remote agent tests, because the compared values must be available on the local machine.

1. Move the mouse cursor over the error in the results file.
2. Click **Results > View Differences**.

You can also click on the red square that precedes the error message in the results file.

The **Difference Viewer** appears, listing every expected (baseline) value in the left pane and the corresponding actual value in the right pane. Differences are marked with red, blue, or green lines, which denote different types of differences, for example deleted, changed, and added items.

3. Compare the values in the **Expected value** and **Actual value** fields.
4. If the error was caused by a wrong expected value, click **Results > Update Expected Value** to modify the expected value in the test case.
5. Click **Results > Next Result Difference** in the menu to find the next difference.



Note: If the error message on which you click relates to the appearance of the application under test, for example when bitmaps have different sizes, Silk Test Classic opens the **Bitmap Tool** to compare the baseline and results bitmaps.

Testplan Pass/Fail Report and Chart

A **Pass/Fail** report lists the number and percentage of tests that have passed during a given execution of the testplan. The report can be subtotaled by an attribute, for example, by Developer.

After you generate a **Pass/Fail** report, you can take these actions:

- Print the report.
- Export the report to a comma-delimited ASCII file.
- Chart a generated Pass/Fail report—that is, produce report information as a graph—or you can directly graph the testplan results information without a preexisting report.

You can mark manual tests as having passed or failed in the **Update Manual Tests** dialog. The **Pass/Fail** report includes in its statistics the manual tests that you have documented as having passed or failed.

Merging Test Plan Results

You can use the **Merge Results** dialog box to merge another result into the active test-plan result file.

A results file consists of a series of results sets, one set for each test plan run. Merging such results sets is useful when:

- Sections of the test plan are run separately, either by one person or by several people, and you need to create a single report on the testing process. For example, let's say that yesterday you ran a section of the test plan consisting of 20 tests and today you ran a different section of the test plan consisting of 10 tests. The merged results set would have today's date and would consist of the results of 30 tests.
- A newer version of a test plan is updated with new tests or sub-plans and you want a single results set to reflect the execution of the original test plan along with the additional tests or sub-plans. For example, if yesterday you ran a test plan consisting of 20 tests and another test plan with 10 tests, and today you

added the second test plan to the first test plan as a sub-plan, the merged results set of yesterday's runs would have yesterday's date and would consist of the results of 30 tests.

1. Click **Results > Merge**.



Note: This command is only available in the menu when a results file produced by running a test-plan is active in the editor.

The **Merge Results** dialog box appears.

2. Select the test plan result that you want to merge into the test plan results file. The selected result is merged into the active test plan results file. The date and time of the altered results set reflect the more recent test run.

How Does Silk Test Classic Synchronize Tests?

This functionality is supported only if you are using the Open Agent.

Many unexpected test failures are related to synchronization issues. Weak synchronization during test replay might generate false negative results, making it difficult to detect actual application problems. Synchronization errors are timing issues that heavily depend on the test environment, making these errors difficult to reproduce and solve. For example, a synchronization error that occurs in a specific test environment might never be reproduced in the development environment. Weak synchronization is one of the most common reasons for an automation project to get unmanageable with a growing set of automated tests.

Silk Test Classic provides automated *test synchronization* for all supported technologies, enabling you to build robust and manageable test sets. During the replay of a test, Silk Test Classic ensures that the test always waits until the AUT is ready to perform the next action. For a verification step in a test, Silk Test Classic ensures that any preceding actions in the test are completed before performing the verification.

To adapt your tests to the specific behavior of your AUT, you can change the values of the following synchronization timeouts:

Synchronization timeout (OPT_SYNC_TIMEOUT)

The maximum time in seconds that Silk Test Classic waits for the AUT to become ready during playback. The default value is 300 seconds.

Window timeout (OPT_WINDOW_TIMEOUT)

The maximum time in seconds that the `Find` method searches for a window. The default value is 5 seconds.

Window retry interval (OPT_WINDOW_RETRY)

If Silk Test Classic cannot immediately find a window, Silk Test Classic will retry to find the window until the window timeout expires. The window retry interval specifies the time in seconds that Silk Test Classic waits before retrying to find the window. The default value is 0.5 seconds.

Timeout to wait for an enabled window (OPT_OBJECT_ENABLED_TIMEOUT)

The maximum time in seconds that Silk Test Classic waits for a window to become enabled during playback. The default value is 1 second.



Note: The timeouts do not overlap.

For detailed information about the automated synchronization that Silk Test Classic provides specifically for Web applications, see [Page Synchronization for xBrowser](#). For detailed information about the synchronization that Silk Test Classic provides specifically for Ajax applications, see [How to Synchronize Test Automation Scripts for Ajax Applications](#).

In addition to the automated synchronization, Silk Test Classic also enables you to manually add wait functions to your scripts. Silk Test Classic provides the following wait functions for manual synchronization:

WaitForObject	Waits for an object that matches the specified locator. Works with an XPath locator or an object map identifier.
WaitForProperty	Waits until the property specified by the <code>propertyName</code> parameter gets the value specified by the <code>expectedValue</code> parameter or until the timeout is reached.
WaitForDisappearance	Waits until the object does not exist or until the timeout is reached.
WaitForChildDisappearance	Waits until the child object specified by the <code>locator</code> parameter does not exist or until the timeout is reached.

If a test randomly fails with an `ObjectNotFoundException`, increase the Window timeout, for example to 30 seconds. For very specific long running operations, for example a click on an object that displays after a long calculation with a progress dialog, manually add the `WaitForObject` method to the test script, to wait until the object is found, or add the `WaitForDisappearance` method to the test script, to wait until the progress dialog is no longer displayed.

Automated synchronization example

Consider the following code sample, where Silk Test Classic tries to click on a button with the caption **Ok**:

```
WINDOW button = Desktop.Find( "//PushButton[@caption='ok' ] ")
button.Click()
```

To replay the actions in this code sample, Silk Test Classic performs the following synchronization actions:

1. Silk Test Classic tries to find the button. If the Window timeout runs out, Silk Test Classic stops the replay and throws an exception.
2. Silk Test Classic waits until the application under test (AUT) is ready. If the Synchronization timeout runs out before the AUT is ready, Silk Test Classic stops the replay and throws an exception.
3. Silk Test Classic waits until the button is enabled. If the Timeout to wait for an enabled window runs out before the button is enabled, Silk Test Classic stops the replay and throws an exception.
4. Silk Test Classic clicks the button.
5. Silk Test Classic waits until the application under test (AUT) is ready again.

Selecting which Results to Display

By default, Silk Test Classic saves the results of the last five executions of a script, suite, or test plan. You can use the **Select Results** dialog box to select which of these results to display.

Click **Results > Select**.



Note: To change the default number of results that are saved, click **Options > Runtime** to open the **Runtime Options** dialog box and edit the **History Size** option.

1. Click **Results > Select**.



Note: The **Results** menu is only available when the active window is a results window.

2. Select the result that you want to display from the **Results to view** list.
3. *Optional:* Add a comment to the selected result by typing the comment into the **Comment** field.
4. Click **OK**.

Select Results to Chart Dialog Box

Use the **Select Results to Chart** dialog box to choose the set of results for which you would like to generate a chart. This dialog is only available when the **Chart** tab is selected on the **Results Pass/Fail Report** dialog box.

Click **Select** on the **Results Pass/Fail Report** dialog.

Export Results Dialog Box

Use the **Export Results** dialog box to export your results to a structured file that is suitable for further processing by an application, such as a spreadsheet. This is only available when the active window is a results window.

Click **Results > Export**.

- Filename area** Displays the path and name of the file to be created. The default path is based on the current directory and the default file name is based on the name of the `.res` file. Click **Browse** to open the **Export Results File Name** dialog box if you need help choosing a new path or name.
- Fields to export area** Specifies all the fields available for export. Select the fields you want to export.
- Export format area** Specifies how the fields are delimited in the file. Select a built-in delimited style in the **Export Format** list or select **Custom** and specify your own delimiters. If you select custom specify your delimiters in the appropriate boxes: **Delimiter**, **Quote**, **Escape delimiter**, and **Escape quote**. Default is **Comma Delimited, Quoted Strings**.
- Write header** If checked, the following header information is included in the file: name of the results file, which fields were exported, and how the fields were delimited. Default is checked.
- Write paths relative to results file** If checked, the name of the directory and file that stores the results file is included in the file. Silk Test Classic always assigns the extension `.res` to all results files.
- Results to export area** Specifies which results to export. Click to select the results you want exported. To select more than one set, press the `Ctrl` key, then click each file you want to export. Default is the set currently displayed in the results window.

View Options Dialog Box

Use the **View Options** dialog box to specify which information you want displayed in the results window.

Click **Results > View Options**.

You must check the **Log elapsed time, thread, and machine for each output line** check box in the **Runtime** dialog box in order to use these options. Click **Options > Runtime** to set the option.

- Display options area** select the information you want to display in the results window. You can select Elapsed time, Thread number, or Current machine.
- Sort lines by area** Select how you want the results sorted. You can specify **Elapsed time**, **Thread number**, or **Current machine**.

When you click **OK**, the current results file displays the options that you specified.

Results Pass/Fail Report Dialog Box

Use the **Results Pass/Fail Report** dialog to generate an on-screen report on the number and percentage of tests that have passed. Available only when a results file produced by running a testplan is the active window.

Click **Results > Pass/Fail Report**.

Subtotal by attribute list box	Select the attribute you want to report/chart by. Default is none.
Report tab	Use this tab to generate a report in tabular format.
Chart tab	Use this tab to generate a report in graphic format. Double-click or right-click the area of the chart that you want to customize, for example, the text that appears for the title and footnote.
Print button	Click to print a chart or report. Opens one of the following dialogs as appropriate: Print Pass/Fail Chart or Print Pass/Fail Report . Available only after you click Generate .
Generate	Button click to generate a chart or report.
Export	Button click to export a Pass/Fail Report as a text file. Available only when Report tab is selected.
Select	Button click to open the Select Results to Chart dialog. Available only when chart tab is selected.
Display percentages	Check to have the chart list the percentage that has passed. Default is unchecked, which displays the number passed. This button is available only when the Chart tab is selected. If you select Display percentages and then uncheck the check box, all colors of the chart change to green. To restore the original colors you must close this dialog and reopen it.
Setup	Displays the Chart Settings dialog. This button is available only when chart tab is selected.

Compare Two Results Dialog Box

Use to see results that have changed from a previous run of the testplan. Available only when a results file produced by running a testplan is the active window.

Click **Results > Compare Two Results**.

Analyzing Results with the Silk TrueLog Explorer

This section describes how you can analyze results with the Silk TrueLog Explorer (TrueLog Explorer).

For additional information about TrueLog Explorer, refer to the *Silk TrueLog Explorer User Guide*, located in (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Documentation** or (in Microsoft Windows 10) **Start > Silk**.

TrueLog Explorer

The TrueLog Explorer helps you analyze test results files and can capture screenshots before and after each action, and when an error occurs. TrueLog Explorer writes the test result files and screenshots into a TrueLog file.

You can additionally use the **Difference Viewer** to analyze results for test cases that use the Open Agent.

You can enable or disable TrueLog Explorer:

- For all test cases using the **TrueLog Options** dialog box.
- Each time you run a specific test case using the **Run Testcase** dialog box.
- At runtime using the test script.

When you enable or disable TrueLog Explorer in the **Run Testcase** dialog box, Silk Test Classic makes the same change in the **TrueLog Options** dialog box. Likewise, when you enable or disable TrueLog Explorer in the **TrueLog Options** dialog box, Silk Test Classic makes the same change in the **Run Testcase** dialog box.



Note: By default, TrueLog Explorer is enabled when you are using the Open Agent, and disabled when you are using the Classic Agent. When TrueLog Explorer is enabled, the default setting is that screenshots are only created when an error occurs in the script and only test cases with errors are logged.

For additional information about TrueLog Explorer, refer to the *Silk TrueLog Explorer User Guide*, located in (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Documentation** or (in Microsoft Windows 10) **Start > Silk**.

TrueLog Limitations and Prerequisites

When you are using TrueLog with Silk Test Classic, the following limitations and prerequisites apply:

Remote agents	When you are using a remote agent, the TrueLog file is also written on the remote machine.
Suites	TrueLog is not supported when you are executing suites.
Mixed-agent scripts	TrueLog is not supported when you are executing mixed-agent scripts, which are scripts that are using both agents.
Multiple-agent scripts	TrueLog is supported only for one local or remote agent in a script. When you are using a remote agent, the TrueLog file is also written on the remote machine.
Open Agent scripts	To use TrueLog Explorer with Open Agent scripts, set the default agent in the toolbar to the Open Agent.
Classic Agent scripts	To use TrueLog Explorer with Classic Agent scripts, set the default agent in the toolbar to the Classic Agent.

Why is TrueLog Not Displaying Non-ASCII Characters Correctly?

TrueLog Explorer is a MBCS-based application, meaning that to be displayed correctly, every string must be encoded in MBCS format. When TrueLog Explorer visualizes and customizes data, many string conversion operations may be involved before the data is displayed.

Sometimes when testing UTF-8 encoded Web sites, data containing characters cannot be converted to the active Windows system code page. In such cases, TrueLog Explorer will replace the non-convertible characters, which are the non-ASCII characters, with a configurable replacement character, which usually is '?'.

To enable TrueLog Explorer to accurately display non-ASCII characters, set the system code page to the appropriate language, for example Japanese.

Opening the TrueLog Options Dialog Box


Use the TrueLog options to enable the TrueLog Explorer and to customize the test result information that TrueLog collects.

- To open the **TrueLog Options** dialog box from the main menu, click **Options > TrueLog**.
- To open the **TrueLog Options** dialog box from a test case, click **Run Testcase** on the **Basic Workflow** bar. If the workflow bar is not visible, click **Workflows > Basic** to enable it. In the **Run Testcase** dialog box, check the **Enable TrueLog** check box and then click **TrueLog Options**.

Setting TrueLog Options

Use the TrueLog options to enable TrueLog and to customize the test result information that the TrueLog collects.

Logging bitmaps and controls in a TrueLog may adversely affect performance. Because capturing bitmaps and logging information can result in large TrueLog files, you may want to log test cases with errors only and then adjust the TrueLog options for test cases where more information is needed.

 **Note:** To reduce the size of TrueLog files with Silk Test 17.5 or later, the file format for TrueLog files with the Open Agent has changed from `.xlg` to the compressed `.tlz` file format. Files with a `.xlg` suffix are automatically appended with a `.tlz` suffix. To parse result data from a `.tlz` file, you can unzip the `.tlz` file and parse the data from the included `.xlg` file. For the Classic Agent, the `.xlg` file format is used.


1. Click **Options > TrueLog** to open the **TrueLog Options** dialog box.
2. To capture TrueLog data and activate logging settings, check the **Enable TrueLog** check box and then choose to capture data for:

All testcases Logs activity for all test cases, both successful and failed. This setting may result in large TrueLog files.

Testcases with errors Logs activity only for test cases with errors. This is the default setting.

3. In the **TrueLog location** field, type the name of and optionally the path to the TrueLog file, or click **Browse** and select the file.

The path is relative to the machine on which the agent is running. The default path is the path of the Silk Test Classic project folder, and the default name is the name of the suite class, with a `.tlz` suffix. To ensure that TrueLog files are not overwritten, for example when you perform parallel testing, you can add placeholders to the TrueLog file name. These placeholders are replaced with the appropriate data at execution time.

 **Note:** The path is validated at execution time. Tests that are executed by Silk Central set this value to the Silk Central results directory to enable the screenshots to be shown in the result view.

4. Only when you are using the Classic Agent, choose one of the following to set pre-determined logging levels in the **TrueLog Presets** section:

Minimal Enables bitmap capture of desktop on error; does not log any actions.

Default Enables bitmap capture of window on error; logs data for Select and SetText actions; enables bitmap capture for Select and SetText actions.

Full Logs all control information; logs all events for browsers except for MouseMove events; enables bitmap capture of the window on error; captures bitmaps for all actions.

If you enable Full logs and encounter a `Window Not Found` error, you may need to manually edit your script.

5. Only when you are using the Classic Agent, in the **Log the following for controls** section, specify the types of information about the controls on the active window or page to log.

6. Only when you are using the Classic Agent, in the **Log the following for browsers** section, specify the browser events that you want to capture.
7. Specify the amount of time you want to allow Windows to draw the application window before a bitmap is taken.
 - When you are using the Classic Agent, specify the delay in the **TrueLog Delay** field.
 - When you are using the Open Agent, specify the delay in the **Delay** field in the **Screenshot mode** section.

The delay can be used for browser testing. You can insert a `Browser.WaitForReady` call in your script to ensure that the `DocumentComplete` events are seen and processed. If `WindowActive` nodes are missing from the TrueLog, you need to add a `Browser.WaitForReady` call. You can also use the delay to optimize script performance. Set the delay as small as possible to get the correct behavior and have the smallest impact on script execution time. The default setting is 0.
8. To capture screenshots of the application under test:
 - When you are using the Classic Agent, check the **Enable Bitmap Capture** check box and then choose to capture bitmaps.
 - When you are using the Open Agent, determine how Silk Test Classic captures screenshots in the **Screenshot mode** section.
9. Only when you are using the Classic Agent, click the **Action Settings** tab to select the scripted actions you want to include in the TrueLog.

When enabled, these actions appear as nodes in the Tree List view of the TrueLog.
10. Only when you are using the Classic Agent, in the **Select Actions to Log** section, check the **Enable** check box to include the corresponding 4Test action in the log. Each action corresponds to a 4Test method, except for `Click` and `Select`.
11. Only when you are using the Classic Agent, in the **Select Actions to Log** section, from the **Bitmap** list box, select the point in time that you want bitmaps to be captured.
12. Click **OK**.

Toggle TrueLog at Runtime Using a Script

This functionality is supported only if you are using the Classic Agent.

Toggle the TrueLog Explorer at runtime to analyze test results, capture screen-shots before and after each action, and capture screen-shots when an error occurs.

Use the test script to toggle TrueLog Explorer multiple times during the execution of a test case. For example, if you run a single test case to test multiple user interface menus, you can turn TrueLog on and off several times during the script to capture bitmaps for only a portion of the menus.

1. Set the TrueLog Explorer options to define what you want the TrueLog Explorer to capture.
2. Create or open the script that you want to modify.
3. Navigate to the portion of the script that you want to turn on or off.
4. To turn TrueLog off, type: `SetOption(OPT_PAUSE_TRUELOG, TRUE)`.
5. To turn TrueLog on, type: `SetOption(OPT_PAUSE_TRUELOG, FALSE)`.
6. Click **File > Save** to save the script.

Viewing Results Using the TrueLog Explorer

Use the TrueLog Explorer to analyze test results files, capture screenshots before and after each action, and capture screenshots upon error.

1. Set the TrueLog Explorer options.

2. Run a test case.
3. Choose one of the following:
 - Click **Results > Launch TrueLog Explorer**.
 - Click the **Explore Results** button on the **Basic Workflow** or the **Data Driven Workflow** bars.
4. On the **Results Files** dialog box navigate to the file name that you want to review and click **Open**.

By default, the results file has the same name as the executed script, suite, or test plan. To review a file in the **TrueLog Explorer**, open a `.tlz` file. To review a Silk Test Classic results file in Silk Test Classic, open a `.res` file.

Modifying Your Script to Resolve Window Not Found Exceptions When Using TrueLog

This functionality is supported only if you are using the Classic Agent.

When you run a script and get a `window 'name' was not found` error, you can modify your script to resolve the issue. Use this procedure if all of the following options are set in the **TrueLog Options - Classic Agent** dialog box:

- The action **PressKeys** is enabled.
- Bitmaps are captured after or before and after the **PressKeys** action.
- **PressKeys** actions are logged.

The preceding settings are set by default if you select **Full** as the TrueLog preset.

To resolve this error, in your test case, use `FlushEvents()` after a `PressKeys()` and `ReleaseKeys()` pair. Or, you can use `TypeKeys()` instead.

There is no need to add `sleep()` calls in the script or to change timeouts.

```
testcase one()
  Browser.SetActive()
  // Google.PressKeys("<ALT-T>")
  // Google.ReleaseKeys("<ALT-T>")
  Google.TypeKeys("<ALT-T>")
  Agent.FlushEvents()
  Google.TypeKeys("O")
  Agent.FlushEvents()

  //recording
  IE_Options.SetActive()
  IE_Options.PageList.Select("Security")
  IE_Options.Security.SecurityLevelIndicator.SetPosition(2)
  BrowserMessage.SetActive()
  BrowserMessage.OK.Click()
  IE_Options.SetActive()
  IE_Options.OK.Click()
```

Analyzing Bitmaps

This section describes how you can analyze bitmaps with the **Bitmap Tool**.

Overview of the Bitmap Tool

This topic contains a brief overview of the **Bitmap Tool**. To access more information about the **Bitmap Tool**, launch it and press `F1` or choose **Help > Help Topics**.

The **Bitmap Tool** is an application that allows you to test and correct your Windows application's appearance by comparing two or more bitmaps and identifying the differences between them. It is

especially useful for testing inherently graphical applications, like drawing programs, but you can also check the graphical elements of other applications. For example, you might want to compare the fonts you expect to see in a dialog with the fonts actually displayed, or you might want to verify that the pictures in toolbar buttons have not changed.

It can be used as a stand-alone product, in which you create and compare bitmaps of entire windows, client areas, the desktop, or selected areas of the screen. More commonly, however, you use the tool in conjunction with Silk Test Classic. Bitmaps captured can be opened in the **Bitmap Tool** where you can compare them using the tool's comparison features. Conversely, bitmaps captured by the bitmap tool can be compared by Silk Test Classic bitmap functions.

You can compare a baseline bitmap captured in the **Bitmap Tool** with one captured in a Silk Test Classic test case of your application.

- If you write test cases by hand, you can use Silk Test Classic built-in bitmap functions.
- If you prefer to record test cases through **Record > Testcase**, the **Verify Window** dialog box allows you to record a bitmap-related verification statement.

The **Bitmap Tool** can only recognize an operating system's native windows. In the case of the Abstract Windowing Toolkit (AWT), included with Sun Microsystems Java Development Kit (JDK), each control has its own window, since AWT controls are native Microsoft windows. As a result, the **Bitmap Tool** will only see the top level dialog box.

When to use the Bitmap Tool

You might want to use the **Bitmap Tool** in these situations:

- To compare a baseline bitmap against a bitmap generated during testing.
- To compare two bitmaps from a failed test.

For example, suppose during your first round of testing you create a bitmap using one of Silk Test Classic's built-in bitmap functions, `CaptureBITmap`. Assume that a second round of testing generates another bitmap, which your test script compares to the first. If the testcase fails, Silk Test Classic raises an exception but cannot specifically identify the ways in which the two images differ. At this point, you can open the **Bitmap Tool** from the results file to inspect both bitmaps.

Capturing Bitmaps with the Bitmap Tool

You can capture bitmaps by embedding bitmap functions and methods in a test case or by using the **Bitmap Tool**. This section explains how to capture bitmaps in the **Bitmap Tool**.

Use the **Capture** menu to capture a bitmap for any of the following in your application:

- A window.
- The client area of a window, which means the working area, without borders or controls.
- A selected rectangular area of the screen. This is especially useful for capturing controls within a window.
- The desktop.

Capturing a Bitmap with the Bitmap Tool

1. Start the application in which you want to capture bitmaps and set up the window or area to capture.
2. Start the **Bitmap Tool**.
3. If you want to change the current behavior of the tool window, click **Capture > Hide Window on Capture**.

By default, the tool window is hidden during capture.

4. Choose a window or screen area to capture:

- Window** Choose **Capture > Window**. Click the window you want to capture.
- Client area** Choose **Capture > Client Area**. Click the client area you want to capture.
- Selected rectangular area** Choose **Capture > Rectangle**.
 1. Move the mouse cursor to desired location to begin capture.
 2. While pressing and holding the left mouse button, drag the mouse to outline a rectangle, and then release the mouse button to capture it. During outlining, the size of the rectangle is shown in pixels.
- Desktop** Click **Capture > Desktop**.

The **Bitmap Tool** creates a new MDI child window containing the newly captured bitmap. The title bar reads **Bitmap - (Untitled)** and the status line at the bottom right of the window gives the dimensions of the bitmap (height by width), and the number of colors.

5. Repeat steps 3 and 4 to capture another bitmap. Alternatively, open an existing bitmap file.
6. Save the bitmap.

Now you are ready to compare the two bitmaps or create a mask for the baseline bitmap.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Capturing a Bitmap During Recording

1. Open the dialog box by pointing at the object you want to capture and pressing `Ctrl+Alt`.
2. Click the **Bitmap** tab.
3. Enter a file name in the **Bitmap File Name** field. Use the **Browse** button to select a directory name. The default path is based on the current directory. The default file name for the first bitmap is `bitmap.bmp`. Click **Browse** if you need help choosing a new path or name.
4. Choose whether to copy the **Entire Window**, **Client Area of Window**, or **Portion of Window**, and click **OK**.

To capture a portion of the window, move the mouse cursor to the location where you want to begin. While pressing the left mouse button, drag the mouse to outline a rectangle, and then release the mouse button to capture the bitmap.

Silk Test Classic always adds a bitmap footer to the bitmap file. This means that the physical size of the bitmap will be slightly bigger than if you capture the bitmap in the **Bitmap Tool**. The bitmap footer always contains the window tag for a given bitmap.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Capturing All or Part of the Zoom Window in Scan Mode

1. Make sure the **Capture > Hide Window** is unchecked. If necessary, select the item to uncheck the check mark.

- Click **Next** or **Previous** until the **Zoom** window contains the difference you want to capture.
- Perform one of the following actions to capture the desired part of the **Zoom** window:

- | | |
|-------------------------------------|---|
| Entire Zoom window | Press <code>Ctrl+W</code> and select the Zoom window. |
| Client area of Zoom window | Press <code>Ctrl+A</code> and select the Zoom window. |
| Selected area of Zoom window | Press <code>Ctrl+R</code> . Move the mouse cursor to desired location to begin capture. While pressing and holding the left mouse button, drag the mouse to the screen location to end capture, and release the mouse button. |

- Optionally, you can fit the bitmap in its window, resize it, and save it.

Saving Captured Bitmaps

You can, if you want, save the bitmaps you have captured in the **Bitmap Tool**. You should adopt a naming convention that helps you distinguish between the first bitmap in the comparison, called the baseline bitmap, and the second bitmap, called the result bitmap. You can make the distinction in the file name itself, for example, by appending or prefixing a `b` or `r` to the name and using the same file extension for all bitmap files. Or you might use the same file name for both baseline and result bitmaps and add a unique file extension.

Example

You save baseline and result bitmaps of the **Open** dialog box as `open.bmp` and `open.rmp`. Alternatively, you might name them `openbase.bmp` and `openres.bmp`, respectively.

The following table lists the file extensions supported by the **Bitmap Tool**. We recommend that you use `.bmp` for baseline bitmaps and `.rmp` for result bitmaps.

If you are saving	And you want the file name to be	Then use this extension
Baseline bitmap	Identical to the result bitmap's	<code>.bmp</code>
Result bitmap	Identical to the baseline bitmap's	<code>.rmp</code>
Either baseline or result bitmap	Unique	<code>.bmp</code> or <code>.dib</code> (Device Independent Bitmap)



Note: Silk Test Classic uses `.rmp` for bitmaps that are captured within a test case and fail verification.

Comparing Bitmaps

The **Bitmap Tool** can create and graphically locate the differences between two bitmaps. You can use all Windows functionality to resize, save, and otherwise manipulate bitmaps, in addition to the special comparison features included in the tool.

Using the **Bitmap Tool**, you can:

- Show the areas of difference.
- Zoom in on the differences.
- Jump from one zoomed difference to the next.
- View on-line statistics about the bitmaps.
- Edit (copy and paste), print, and save bitmaps.
- Create masks.

The **Bitmap Tool** has the following major comparison commands:

Command	Description
Show	Creates a Differences window, which is a child window containing a black-and-white bitmap. Black represents areas with no differences and white represents areas with differences.
Zoom	<p>Creates a special, not sizable, Zoom window with three panes and resizes and stacks the Baseline, Differences, and Result windows.</p> <ul style="list-style-type: none"> • The top pane of the Zoom window contains a zoomed portion of the Baseline window. • The middle pane shows a zoomed portion of the Differences window. • The bottom pane shows a zoomed portion of the Result window. <p>All three zoomed portions show the same part of the bitmap. When you move the mouse within any of the three windows, the Bitmap Tool generates a simultaneous and synchronized real-time display in all three panes of the Zoom window.</p> <p>While in scan mode, you can capture the Zoom window to examine a specific bitmap difference.</p>
Scan	The tool indicates the location of the first difference it finds by placing a square in the same relative location of the Baseline , Result , and Differences windows. The three panes of the Zoom window also show the difference.
Comparison Statistics	Provides statistics about the bitmaps.

You can also compare bitmaps by creating and applying masks.

Rules for Using Comparison Commands

You should be familiar with the following rules before using the commands:

- If you are comparing two new bitmaps captured in the tool, designate one bitmap as the baseline, the other as the result bitmap.
- If you are comparing two existing, saved bitmaps, open first the bitmap that you consider the baseline. The tool automatically designates the first bitmap you open as the baseline, and the second as the result.
- The commands must be used in this order: **Show**, **Zoom**, and **Scan**.

Bitmap Functions

`CaptureBitmap`, `SYS_CompareBitmap`, `WaitBitmap`, and `VerifyBitmap` are built-in bitmap-related 4Test functions. In particular, `VerifyBitmap` is useful for comparing a screen image during the execution of a test case to a baseline bitmap created in the **Bitmap Tool**. If the comparison fails, Silk Test Classic saves the actual bitmap in a file. In the following example, the code compares the test case bitmap (the baseline) against the bitmap of `TestApp` captured by `VerifyBitmap`:

```
TestApp.VerifyBitmap ("c:\sample\testbase.bmp")
```

Baseline and Result Bitmaps

To compare two bitmaps, you must designate one bitmap in the comparison as the baseline and the second bitmap as the result. While you may have many bitmap files open in the **Bitmap Tool**, at any one time only one bitmap can be set as the baseline and one as the result. If you want to set new baseline and result bitmaps, you must first un-set the current assignments.

These designations are temporary and at any time you can set and reset a bitmap as a baseline, result, or neither.

Designating a Bitmap as a Baseline

To designate a bitmap as a baseline:

In the **Bitmap Tool**, click **Bitmap > Set Baseline**. The **Set Baseline** menu item is checked. The title bar of the child window changes to **Baseline Bitmap -- filename.bmp**.

Designating a Bitmap as a Results File

To designate a bitmap as a results file:

In the **Bitmap Tool**, click **Bitmap > Set Result**. The **Set Result** menu item is checked. The title bar of the child window changes to **Result Bitmap -- filename.rmp**.

Un-Setting a Designated Bitmap

Uncheck the menu item. For example, to un-set a baseline bitmap, uncheck **Bitmap > Set Baseline**. The check mark is removed.

Uncheck the menu item.

For example, to un-set a baseline bitmap, uncheck **Bitmap > Set Baseline**.

The check mark is removed.

Zooming the Baseline Bitmap, Result Bitmap, and Differences Window

Choose **Differences > Show** and then **Differences > Zoom**.

The tool arranges the **Baseline Bitmap** on top, the **Result Bitmap** on the bottom, and the **Differences** window in the middle. To the right of these, the tool creates a **Zoom** window with three panes, arranged like the bitmap windows

Looking at Statistics

The **Differences > Comparison Statistics** command displays information about the baseline and result bitmaps, with respect to width, height, colors, bits per pixel, number of pixels, and the number and percentage of differences (in pixels).

Viewing Statistics by Comparing the Baseline Bitmap and the Result Bitmap

To view statistics by comparing the baseline bitmap and the result bitmap:

Click **Differences > Comparison Statistics**. The **Bitmap Comparison Statistics** window opens.



Note: The number of colors is derived from the following formula: number of colors = $2^{(\text{bits per pixel})}$.

Exiting from Scan Mode

To exit from the scan mode:

Click **Differences > Scan**. Exiting scan leaves the tool in zoom mode.

Starting the Bitmap Tool

This section lists the locations from which you can start the **Bitmap Tool**.

Starting the Bitmap Tool from its Icon and Opening Bitmap Files


1. Click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Tools > Silk Test Bitmap Tool** or (in Microsoft Windows 10) **Start > Silk > Silk Test Bitmap Tool**. The **Bitmap Tool** window displays.
2. Do one of the following:

Open an existing bitmap file Click **File > Open** and specify a file in the **Open** dialog box. See *Overview of Comparing Bitmaps*.

Capture a new bitmap See *Capturing a Bitmap in the Bitmap Tool*.

Starting the Bitmap Tool from the Results File

When the verification of a bitmap fails in a test case, Silk Test Classic saves the actual result in a bitmap file with the same name as the baseline bitmap but with the extension `.rmp`. So, if the bitmap file `testbase.bmp` fails the comparison, Silk Test Classic names the result bitmap file `testbase.rmp`. It also logs an error message in the results file.

 **Note:** In some cases this error message does not reflect an actual error. In particular, when Silk Test Classic compares a bitmap it captured with one captured in the **Bitmap Tool**, the comparison fails because Silk Test Classic stores footer information in its bitmap. The bitmaps might in fact be identical in all ways except for this information.

To compare the actual bitmap generated by the test case against the baseline bitmap generated by the bitmap tool or one of Silk Test Classic's built-in functions, click the box icon preceding the error message.

Silk Test Classic opens the bitmap tool, opens both the baseline bitmap, which is the expected bitmap as a `.bmp` file, and the result bitmap, which is the actual bitmap as a `.rmp` file, creates a **Results/View Differences** and places it in between the baseline bitmap and the result bitmap. The right portion of the tool displays a three-paned **Zoom** window.

Starting the Bitmap Tool from the Run Dialog Box

1. Click **Start > Run**. The **Run** dialog box displays.
2. Type the pathname of the tool's executable file and any parameters in the **Command Line** field and click **OK**. The **Bitmap Tool** starts. Any bitmaps you specified on the command line are opened.
3. See *Overview of Comparing bitmaps*.
4. If you did not specify any files in the command line, go to the next step.
You can now open existing bitmaps created in Silk Test Classic or in the tool, or you can capture new bitmaps.
5. Do one of the following:

Open an existing bitmap file Click **File > Open** and specify a file in the **Open** dialog box. See *Overview of Comparing Bitmaps*.

Capture a new bitmap See *Capturing a Bitmap in the Bitmap Tool*.

Using Masks

A mask is a bitmap that you apply to the baseline and result bitmaps in order to exclude any part of a bitmap from comparison by the **Bitmap Tool**. For example, if you are testing a custom object that is painted on the screen and one part of the object is variable, you might want to create a mask to filter out the variable part from the bitmap comparison.

You might consider masking any differences that you decide are insignificant or that you know will vary in an effort to avoid test case failure. For example, suppose a test case fails because one bitmap includes a flashing area of a dialog box. In the **Bitmap Tool** you can block the flashing area from the two bitmaps by creating and applying a mask to them. Once a mask is applied and the masked bitmaps are saved, the mask becomes a permanent part of the baseline bitmaps you are comparing. Masks can also be saved in separate files and used in test cases.

You can create a mask in two ways:

- By converting the **Differences** window to a mask. A mask created this way filters out all differences.
- By opening a new mask window and specifying rectangular areas to mask.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Prerequisites for the Masking Feature

Before using the masking feature, you must:

- Capture or open two bitmaps to compare. Set `baselinesetbaseline` and `resultsetresult` bitmaps, if currently un-set.
- Determine which sections you need to mask. Use one or more comparison `featurescomparisoncmds`, if necessary, to locate bitmap differences.

Applying a Mask

1. Open the mask bitmap file and click **Bitmap > Set Mask**.
2. Click **Edit > Apply Mask**.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Editing an Applied Mask

You can edit a mask after it has been applied:

- To add to the mask, place the mouse cursor in the baseline bitmap window at the position where you want to begin adding to the mask. Click and drag the mouse cursor to outline a rectangle. Then release the left mouse button.
- To delete part of the mask, place the mouse cursor in the baseline bitmap window at the position where you want to begin deleting part of the mask. While pressing and holding the **Shift** key, drag the mouse cursor over the area of the existing map that you want to delete, and then release the **Shift** key and the left mouse button.

Creating and Applying a Mask that Excludes Some Differences or Just Selected Areas

1. Click **Edit > New Mask**. The bitmap tool creates an empty **Mask Bitmap child** window that is the same size as the baseline bitmap.

2. Using the **Differences** window to help you locate differences, place the mouse cursor in the baseline bitmap window at the position where you want to begin creating the mask. As you press and hold the left mouse button, drag the mouse cursor to outline a rectangle. Then release the left mouse button. The rectangular outline in the baseline map changes to a filled-in rectangle. The mask bitmap window also contains a like-sized rectangle in the same relative location.
3. Repeat step the previous step until you have completed the mask.
4. If you want to delete a portion of the mask, place the mouse cursor in the baseline bitmap window at the position where you want to begin editing. While pressing the Shift key and then the left mouse button, drag the mouse cursor over the area of the existing map that you want to delete, and then release the Shift key and the left mouse button.

The area of the mask overlapped by the rectangle outline disappears in both the baseline and mask bitmap window.

5. Choose **Edit > Apply Mask**. The bitmap tool applies the mask to the result bitmap and closes the **Differences** window.
6. Choose one of the following actions:

Keep the baseline and result bitmaps with the mask applied	Save the bitmap files. The mask is now a permanent part of the bitmap files.
Unapply the mask	Close the mask bitmap window. Saving is optional.
Keep the mask as it is	Save the mask file.
Edit the mask	Choose File > Save and close the mask bitmap window. This un-applies the mask.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Creating and Applying a Mask that Excludes All Differences

1. Click **Differences > Show** to open a **Differences** window, if one is not already open.
2. Click **Differences > Convert to Mask**. A message is displayed: `Bitmaps are now identical on screen.`
3. Click **OK**.

The bitmap tool creates an untitled mask bitmap from the **Differences** window, swapping black and white, and applies the mask to the baseline and result bitmaps.

4. Choose one of the following actions:

Keep the baseline and result bitmaps with the mask applied	Save the bitmap files. The mask is now a permanent part of the bitmap files.
Unapply the mask	Close the mask bitmap window. Saving is optional.
Keep the mask as it is	Save the mask file.
Edit the mask	Choose File > Save and close the mask bitmap window. This un-applies the mask.

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you

capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Saving a Mask

Masks can be saved in a file, applied to the baseline and result bitmaps for you to examine on screen only, or applied to and saved in the baseline and result bitmap files. Once masks are applied and saved, they become a permanent part of the baseline and result bitmaps. The advantage of saving the mask alone is that later you can read in the mask file and apply it to the bitmap on screen, thus allowing you to keep the bitmap in its original state.

You can supply the name of a mask bitmap file (as well as its associated baseline bitmap file) as an argument to bitmap functions.

The **Bitmap Tool** supports the `.msk` file extension for mask files. Alternatively, you can designate a mask in the file name and use the generic `.bmp` extension. We recommend, however, that you use the `.msk` extension.

The following bitmap-related functions accept mask files as arguments:

- `GetBitmapCRC`
- `SYS_CompareBitmap`
- `VerifyBitmap`
- `WaitBitmap`

The Open Agent and Classic Agent capture bitmaps in different color depths. By default, the Open Agent captures bitmaps using the current desktop settings for color depth. The Classic Agent captures bitmaps as 24-bit images. If you create a mask file for the captured bitmap using the Bitmap tool, the mask file is saved as a 24-bit bitmap. If the bitmap mask file does not have the same color depth as the bitmap that you capture, an error occurs. To ensure that `VerifyBitmap` functions do not fail due to different color-depth settings between the captured image and the mask image, ensure that the bitmaps have the same color depth.

Analyzing Bitmaps for Differences

This section describes how you can analyze bitmaps for differences.

Scanning Bitmap Differences

To scan the differences between the baseline and result bitmaps:

Click **Differences > Scan** or **Differences > Next**. The tool indicates the location of the first difference it finds by placing a square in the same relative location of the **Baseline**, **Result**, and **Differences** windows. The three panes of the **Zoom** window also show the difference.

Showing Areas of Difference

The **Show** command creates a **Differences** window which is a child window containing a black-and-white bitmap. Black represents areas with no differences and white represents areas with differences.

Graphically Show Areas of Difference Between a Baseline and a Result Bitmap

To graphically show the differences between a baseline and a result bitmap:

Click **Differences > Show**. The **Bitmap Tool** displays a **Differences** window along with the source baseline and result bitmaps from which it was derived.

Moving to the Next or Previous Difference

You must first create a **Differences** window and a **Zoom** window using **Differences > Show** and **Differences > Zoom**.

The **Scan** command on the **Differences** menu automates zoom mode and causes the bitmap tool to scan for differences from left to right and top to bottom. When the first difference is found, a small square, 32 x 32 pixels, is shown in the **Baseline Bitmap**, **Result Bitmap**, and **Differences Bitmap** windows in the same relative location. In addition, that location is shown in all three panes in the **Zoom** window.

Click **Differences > Next** or **Differences > Previous**.

Zooming in on the Differences

The **Zoom** command creates a special, not sizable, **Zoom** window with three panes and resizes and stacks the **Baseline**, **Differences**, and **Result** windows.

- The top pane of the **Zoom** window contains a zoomed portion of the **Baseline Bitmap** window.
- The middle pane shows a zoomed portion of the **Differences** window.
- The bottom pane shows a zoomed portion of the **Result Bitmap** window.

All three zoomed portions show the same part of the bitmap. When you move the mouse within any of the three windows, the bitmap tool generates a simultaneous and synchronized real-time display in all three panes of the **Zoom** window.

While in scan mode, you can capture the **Zoom** window to examine a specific bitmap difference.

Working with Result Files

This section describes how you can use result files to interpret the results of your tests.

Attaching a Comment to a Result Set

You can attach comments to individual result sets to record useful information about the test run:

1. Open the result file.
2. Click **Results > Select** to display the **Select Results** dialog.
3. Select the result set to which you want to attach a comment.
4. Type the comment in the **Comment** text field at the bottom of the dialog. The comment displays in the **Comment** column in the **Select Results** dialog.
5. Click **OK**.

Silk Test Classic displays the comments in the various dialogs that list results sets, such as the **Extract Results** and **Delete Results** dialogs.

Comparing Result Files

The **Compare Two Results** command allows you to quickly note only the results that have changed from a prior run without having to look at the same errors over again. The command identifies differences based on the following criteria:

- A test passes in one test plan run and fails in the other.
- A test fails in both runs but the error is different.
- A test is executed in one test plan run but not in the other.

Silk Test Classic uses the test descriptions as well as the test statements to identify and locate the various cases in the test plan. Therefore, if you change the descriptions or statements between runs, Silk Test Classic will not be able to find the test when you run **Compare Two Results**.

1. Open two results files.
2. Make the results set you want to compare to another results set the active window.
3. Choose **Results > Compare Two Results**.
4. On the **Compare Two Results** dialog, select a results set from the list box and click **OK**.
5. When the results set is displayed again, a colored arrow is positioned in the left margin for every test that is different.

A red arrow indicates that the difference is due to the pass/fail state of the test changing.

A magenta arrow indicates that the difference is due to the addition or removal of the test in the compared test run.

6. Click **Results > Next Result Difference** to search for the next difference or choose **Results > Next Error Difference** to search for the next difference that is due to the change in a pass/fail state of a test.

Silk Test Classic uses the test descriptions as well as the script, testcase, and testdata statements to identify and locate the various cases in the test plan and in the results set. When test results overlap in the two results set that were merged, the more recent run is used. If you change a test description between runs or modify the statements, Silk Test Classic might be unable to find the test when you try to merge results. Silk Test Classic places these orphaned tests at the top of the results set.

Customizing results

You can modify the way that results appear in the results file as follows:

- Change the colors of elements in the results file
- Change the default number of results sets
- Display a different set of results
- Remove the unused space in a results file

You can also view an individual summary.

Deleting Results

You can use the **Delete Results** dialog box to delete results.

1. Click **Results > Delete**.



Note: This menu is only available when a results file is active.

Silk Test Classic displays the **Delete Results** dialog box.

2. Select the result that you want to delete from the list.

The most current results set displayed first.

3. Click **OK**.

Change the default number of result sets

1. Click **Options > Runtime**. The **Runtime Options** dialog box displays.
2. In the **History Size** field, change the number to the number of results files you want.



Note: By default, five result sets are kept.

Changing the Colors of Elements In the Results File

1. In Silk Test Classic, click **Options > Editor Colors** to display the **Editor Colors** dialog.
2. Select an element from the **Editor Item** list box.
3. Select one of the 16 colors from the palette or modify the RGB values of the selected color. To modify RGB value, select the color. Slide the bar to the left or right, click the spin buttons, or type specific RGB values until you get the color you want.
4. When you are satisfied with the color, click **OK**.

To revert to the default colors, click **Reset**. By default, these results file elements are displayed in the following colors:

Results file element	Default color/icon
Error messages and warnings	Red plus sign (bold on black-and-white monitor)
Warnings only	Purple plus sign
Test descriptions of executed tests	Dark blue
Test descriptions of unexecuted tests	Grayed out
Other descriptive lines	Black

Fix incorrect values in a script

1. Make the results file active.
2. Click **Results > Update Expected Value**.
3. Optionally, select **Run > Testcase** in order to run the test and confirm that it now passes. The expected values in the script are replaced with the actual values found at runtime.

Marking Failed Test Cases

When a test plan results file shows test case failures, you might choose to fix and then rerun them one at a time. You might also choose to rerun the failed test cases at a slower pace without debugging them to watch their execution more carefully.

Make the results file active and click **Results > Mark Failures in Plan**.

All failed test cases are marked and the test plan is made the active file.

Merging Test Plan Results

You can use the **Merge Results** dialog box to merge another result into the active test-plan result file.

A results file consists of a series of results sets, one set for each test plan run. Merging such results sets is useful when:

- Sections of the test plan are run separately, either by one person or by several people, and you need to create a single report on the testing process. For example, let's say that yesterday you ran a section of the test plan consisting of 20 tests and today you ran a different section of the test plan consisting of 10 tests. The merged results set would have today's date and would consist of the results of 30 tests.
- A newer version of a test plan is updated with new tests or sub-plans and you want a single results set to reflect the execution of the original test plan along with the additional tests or sub-plans. For example, if yesterday you ran a test plan consisting of 20 tests and another test plan with 10 tests, and today you

added the second test plan to the first test plan as a sub-plan, the merged results set of yesterday's runs would have yesterday's date and would consist of the results of 30 tests.

1. Click **Results > Merge**.



Note: This command is only available in the menu when a results file produced by running a test-plan is active in the editor.

The **Merge Results** dialog box appears.

2. Select the test plan result that you want to merge into the test plan results file. The selected result is merged into the active test plan results file. The date and time of the altered results set reflect the more recent test run.

Navigating to errors

To find and expand the next error or warning message in the results file, choose **Edit > Find Error**. To skip warning messages and find error messages only, in the **Runtime Options** dialog, uncheck the check box labeled **Find Error stops at warnings**.

You can also use the **Find**, **Find Next**, and **Go to Line** commands on the **Edit** menu to navigate through a results file.

To expand an error message to reveal the cause of an error, click the red plus sign preceding the message. In addition to the cause, you can see the call stack which is the list of 4Test functions executing at the time the error occurred.

There are several ways to move from the results file to the actual error in the script:

- Double-click the margin next to an error line to go to the script file that contains the 4Test statement that failed.
- Click an error message and choose **Results > Goto Source**.
- Click an error message and press **Enter**.

To navigate from a testplan test description in a results file to the actual test in the testplan, click the test description and click **Results > Goto Source**.

Viewing an individual summary

1. Click a testcase line in a suite or script results file, or click a test description in a testplan results file.
2. Click **Results > Show Summary**.

Storing and Exporting Results

You can store and export results in the following ways:

- Store results in an unstructured ASCII format.
- Export results to a structured file for further manipulation.

Storing results

Silk Test Classic allows you to extract the information you want in an unstructured ASCII text format and send it to a printer, store it in a file, or look at it in an editor window.

To store results in an unstructured ASCII format

1. Click **Results > Extract**.
2. In the **Extract To group** box on the **Extract Results** dialog, select the radio button for the destination of the extracted output: **Window (default)**, **File**, or **Printer**.

3. In the Include group box, check one or more check boxes indicating which optional text, if any, to extract. (This optional text is in addition to the output selected in the **Expand group** box.) The choices are:
4. Select a radio button in the **Expand** group box indicating which units to extract information about. Select **Scripts**, **Scripts and Testcases (default)**, or **Anything with Errors**.
5. Select one or more results sets from the **Results to Extract** group box.
6. Click **OK**.

Extracting Results

Use the **Extract Results** dialog box to extract results in an unstructured ASCII text format.

1. Click **Results > Extract**. This menu is only available if the active file is a results file.
2. Select the destination of the extracted output.
 - To view the extracted results in a new editor window, select **Window**.
 - To store the extracted results in a new file, select **File**.
 - To send the extracted results to a printer, select **Printer**.

The default setting is **Window**.

3. Select which information you want to extract.
 - To extract only script information, select **Scripts**.
 - To extract script and test case information, select **Scripts and testcases**.
 - To extract all error information, select **Anything with errors**.

The default setting is **Scripts and testcases**.

4. *Optional:* Specify which optional text to extract.
 - Check the **Output text** check box to extract the output of sprint statements.
 - Check the **Error text** check box to extract text generated by the `LogError` and `ExceptLog` functions.
 - Check the **Summary text** check box to extract the description of which tests passed and failed and the number of errors that were produced by failed tests.
5. Select the results that you want to extract from the **Results to extract** list.

To select more than one result, press **Ctrl** and click each file you want to extract. The default result is the result that is currently displayed in the results window.
6. Click **OK**.

Exporting Results

You can use the **Export Results** dialog box to export your results to a structured file that is suitable for further processing by an application, such as a spreadsheet.

- | | |
|---|--|
| Write header | If checked, the following header information is included in the file: name of the results file, which fields were exported, and how the fields were delimited. Default is checked. |
| Write paths relative to results file | If checked, the name of the directory and file that stores the results file is included in the file. Silk Test Classic always assigns the extension <code>.res</code> to all results files. |
| Results to export area | Specifies which results to export. Click to select the results you want exported. To select more than one set, press the <code>Ctrl</code> key, then click each file you want to export. Default is the set currently displayed in the results window. |

1. Click **Results > Export**.

This menu is only available when the active window is a results window.

The **Export Results** dialog appears.

2. Specify the file name.

The default path is based on the current directory and the default file name is based on the name of the `.res` file. To select a file, click **Browse** and navigate to the file in the **Export Results File Name** dialog box.

3. *Optional:* Check the corresponding check boxes in the **Fields to export** area to specify the fields that you want to export to the file.

4. *Optional:* In the **Export format** area, specify how you want the fields delimited in the file.

The default is to comma delimit the fields and to put quotation marks around strings. Select **Custom** to specify your own delimiters.

5. *Optional:* To include header information in the file, check the **Write header** check box.

Header information contains the name of the results file, which fields were exported, and how the fields were delimited.

6. *Optional:* To include the directory and file that stores the results file in the file, check the **Write paths relative to the results file** check box.

7. Select the results that you want to export from the **Results to export** list.

By default, the results set that is currently displayed in the results window is extracted.

8. Click **OK**. The information is saved in a delimited text file. You can import that file into an application that can process delimited files, such as a spreadsheet.

Removing the Unused Space from a Result File

1. Open a result file.

2. Click **Results > Compact**. The file size is reduced.

Logging Elapsed Time, Thread, and Machine Information

Using the **Runtime Options** dialog box, you can specify that you want to log the elapsed time, the thread number, and the current machine information during execution. This information is then written to the results file where you can display and sort it. For example, if you encounter nested test cases in the results files because you use multi-threading, check this check box to record thread number information in your results file. Then, you can sort the lines in your results file by the thread numbers to better navigate within the nested test cases.

1. Click **Options > Runtime** to open the **Runtime Options** dialog box.

2. In the **Results** area, check the **log elapsed time, thread, and machine for each output line** check box.

3. Click **OK**.

Presenting Results

This section describes how you can use charts and reports to present the results of your tests.

Fully customize a chart

1. Generate the **Pass > Fail** report and click the **Chart** tab.

2. Click the area of the chart that you want to customize, for example, the text that appears for the title and footnote.
3. Double-click the selected area. A dialog displays showing the properties for the selected area. (You can also right-click anywhere on a chart and select the area you want to modify from a popup menu.)
4. Make your changes.
5. Click **OK**.

Generate a Pass/Fail Report on the Active Test Plan Results File



Note: You can only generate pass/fail reports for the results of test plans, not for the results of individual tests.

1. Make sure the test plan results file you want to report on is active, and then click **Results > Pass/Fail Report**.
2. On the **Results Pass/Fail Report** dialog box, select an attribute to report on from the **Subtotal by Attribute** list.
3. Click **Generate**.
4. Take one of the following actions:

Subtotal the report by a different attribute

Select a different attribute in the **Subtotal By Attribute** list, then click **Generate**.

Print the report

Click **Print**. You can set the margins, headers and footers, print quality, and fonts for the report. To change the font, click **Font**. To change the printer setup, click **Setup**.

When you have finished setting these options, click **OK** to print the report.

Chart the report

Display the **Chart** tab.

Write the report to a comma-delimited ASCII file

Click **Export**, specify the full path of the file and click **OK**.

You can open the file in a spreadsheet application that accepts comma-delimited data.

Producing a Pass/Fail Chart

You can create a chart out of a generated **Pass/Fail** report, or you can directly create a graph of the test plan results information without a preexisting report.



Note: You can only generate pass/fail reports for the results of test plans, not for the results of individual tests.

1. Open the result file of a test plan execution in Silk Test Classic.
2. In the Silk Test Classic menu, click **Results > Pass/Fail Report**. The **Pass/Fail Report** dialog box opens.
3. Click the **Chart** tab.

If you have already generated a report, Silk Test Classic displays a chart of the generated report. You might need to resize the window so there is enough room to display the chart well. If you have not generated a report, Silk Test Classic displays a default chart, which allows you to modify chart parameters before you actually generate the chart.

4. Perform one of the following actions:

Change basic charting properties

1. Click **Setup**. The **Chart Settings** dialog is displayed.
2. To change the chart type, select an option from the **Chart Type** list. Silk Test Classic provides bar charts, line charts, and area charts.
3. Click **Apply** to update the chart and leave the **Chart Settings** dialog open. You can also choose whether the chart is three-dimensional, is stacked (for bar charts), and displays a legend, which describes the data being charted. Silk Test Classic displays a model that represents how the chart will look based on current settings.

Add the results from another execution of the test plan to the chart

1. Click **Select**. The **Select Results** dialog is displayed, listing recent runs of the current test plan. Silk Test Classic keeps a history of results for each test plan. The number of results it keeps is determined by the value for **History Size** in the **Runtime Options** dialog.
2. Select the results you want to add to the chart. The results from the selected execution of the test plan will be added to the results currently charted. You can use this feature to compare two different runs of the same tests to spot problem areas. You can chart today's results, then click **Setup** and select yesterday's results to have both appear on one chart.

Move a part of the chart

1. Click the part you want to move, such as the title, legend, or footnote (the text that displays below the chart). The area is selected.
2. Drag it with the mouse.

Print the chart

1. Click **Print**. The **Print Pass/Fail Chart** dialog displays. You can specify a header or footer.
2. Click **OK** to print the chart.

Copy the chart to the clipboard

1. Right-click anywhere on the displayed chart, and then click **Copy**.
2. The chart is placed on the clipboard. You can paste it into another application.

Change advanced charting properties

Usually you can get the chart you want using the default and basic charting properties. But if you want more customization, you can modify just about any property in the chart, including:

- text that appears for the title and footnote
- font used for any text in the chart
- location for the title, legend, and footnote
- colors used for the data
- size and spacing of the bars in bar charts
- borders and shading to the background (backdrop) of any area
- See Customizing a chart.

Generate the chart

Once you are satisfied with the chart parameters, click **Generate**. The **Pass/Fail** chart is displayed.

Displaying a different set of results

1. Click **Results > Select**. Silk Test Classic displays the **Select Results** dialog with the most current results set displayed first.
2. Select the set of results you want to see and click **OK**.

Debugging Test Scripts

Errors encountered during playback can be caused by a variety of factors, such as changes in the application under test (AUT), improper test step flow, or environmental changes. Quickly diagnosing and fixing these errors using debugging features minimizes test maintenance and allows a more efficient team testing effort.

Silk Test Classic automatically raises an exception in response to many of errors or inconsistencies in your scripts and your keyword-driven tests. However, some problems might cause a script or a keyword-driven test to work in unexpected ways without generating an exception. To solve such problems, use the debugger.

Debugging enables you to manage, examine, reset, or step through the actions in a script or keyword-driven test. While debugging, you can specify breakpoints to stop the execution before or after a specific action, and you can examine the values of any local and global variables. You can also enter expressions to evaluate. This lets you identify exactly where an error might be occurring.

You can also use the debugger to find problems in your application under test (AUT). Use the debugging facilities to step through the application and to determine the exact location where a problem occurs.

To start debugging, open the test script or the keyword-driven test script in the editor and click **Run > Debug** in the Silk Test Classic menu. You can also click **Debug** in the toolbar.

The debugger enables you to view the results of your testing in the following ways:

- View the debugging transcript when you debug a script. For additional information, see *Viewing the Debugging Transcripts*. Silk Test Classic records the error information and the output of any print statements in the transcript instead of into a results file.
- Examine the debugging variables while you are debugging a test script. For additional information, see *Viewing Variables*.
- View the call stack. The call stack is a description of all function calls that are currently active in the script which you are debugging. By viewing the call stack, you can trace the flow of execution, which might uncover errors generated by an erroneous flow of control in the script. To view the current call stack, choose **View > Call Stack** in the Silk Test Classic menu. Silk Test Classic displays the call stack in a new window. To return to the script being debugged, press **F6** or choose **View > Module** and select the script from the list.

You cannot use the debugger from plan (*.pln) files, however, you could call test cases from a `main()` function and debug the test cases from there.



Note: You cannot modify files while you are using the debugger. To fix a problem in a file, stop the debugger by clicking **Debug > Exit** in the Silk Test Classic menu.

Designing and Testing with Debugging in Mind

Designing and testing a script to facilitate debugging might significantly reduce test maintenance costs. The following suggestions might help you create debuggable scripts:

- Plan for debugging and robustness when you are designing a script, by having your functions check for valid input and output, and by performing operations in the script that inform you if a problem occurs.
- Test each function while you write it, by building it into a small script that calls the function with test arguments and performs some functional validation. When you have finished the coding of a script, you can use the debugger to step through the execution of each function in the script.
- To find errors in control loops, test each routine with the full range of valid data values, including the highest and lowest valid values.

- Test each routine with invalid values to ensure that the routine rejects such values without crashing.
- Test each routine with null (empty) values. Depending on the purpose of the script, it might be useful to provide default values if the input is incomplete.

Executing a Script in the Debugger

Once you have set one or more breakpoints in your test script or your keyword-driven test script, perform the following actions to execute the script:

1. In the Silk Test Classic menu, click **Debug > Run**. Silk Test Classic executes the script until it reaches the line in which the first breakpoint is located, until an error occurs, or until the script ends. A blue arrow marks the line where Silk Test Classic stopped running the script.
2. Click **Debug > Continue**. Silk Test Classic executes the script until it reaches the line in which the first breakpoint is located, until an error occurs, or until the script ends.
3. Perform the following actions as required:
 - Click **Debug > Step Into** to execute the current line in the active script or in a file that is called by the active script. This action is available only if the execution has stopped at a breakpoint. If the current line contains a function call, control passes into the function and the execution stops at the first statement in the function.
 - Click **Debug > Step Over** to execute the current line of code in the active script or in a file that is called by the active script, without stepping into any functions called by the current line. The execution stops at the next statement. This action is available only if the execution has stopped at a breakpoint.
 - Click **Debug > Run to Cursor** to execute the script until the line containing the cursor.
 - Click **Debug > Finish Function** to execute the script until the current function returns.

Debugging a Test Script

Start the debugger to debug your test scripts.



Note: You cannot modify files while you are using the debugger. To fix a problem in a file, stop the debugger by clicking **Debug > Exit** in the Silk Test Classic menu.

1. To debug a script that is not currently active in the editor, click **File > Debug**.
 - To debug the active script, click **Run > Debug**. Silk Test Classic enters the debugger and pauses without setting a breakpoint.
 - To debug a specific test case from the active script, click **Run > Testcase**. Then select a test case from the **Run Testcase** dialog, and click **Debug**. Silk Test Classic enters the debugger and sets a breakpoint at the first line of the test case.
 - To debug an application state, click **Run > Application State**. Then select an application state from the **Run Application State** dialog box, and click **Debug**. Silk Test Classic enters the debugger and sets a breakpoint at the first line of the application state definition.
 - To debug a plan file, call the test cases in the plan file from a `main()` function and debug the plan file from there. You cannot use the debugger from plan files (*.pln).
2. If you want to debug a script that is not currently active in the editor, select the script file from the **Debug** dialog box. Silk Test Classic enters the debugger and pauses without setting a breakpoint.
3. Click **Open**. Silk Test Classic performs the following actions:
 - Opens the selected script file in debugging mode.
 - Marks the current line, which is the next line to be executed, with a triangle.
 - Changes the menu item from **Run** to **Continue**.

- When script execution completes, a message box displays indicating that the script has terminated.

During script execution, Silk Test Classic displays a transcript window, which is similar to the results window. Unlike the results file, however, the output from debugging a script is not saved in a file, there are no statistics, and all the information is expanded automatically. The transcript window contains the script name, the test case names, and a list of the errors encountered and their line numbers. At the bottom of the transcript window is a text field in which you can enter any statement to execute. The results of each statement you execute appear in the transcript window.

Special debugging commands are available.

Debugger Menus

In debugging mode, the menu bar includes the following three additional menus:

- The commands in the **Debug** menu enable you to control the flow of the script.
- The commands in the **Breakpoint** menu enable you to add or remove breakpoints.
- The commands in the **View** menu enable you to view various elements of the running script, for example local and global variables, the call stack, and breakpoints, and to evaluate expressions.

Stepping Into and Over Functions

To locate a bug in your code you can divide the script into discrete functions and debug each function separately. One way to do this is with the following commands on the **Debug** menu:

Step Into	Step through the function one line at a time, executing the current line in the active script or in a file that is called by the active script. This action is available only if the execution has stopped at a breakpoint. If the current line contains a function call, control passes into the function and the execution stops at the first statement in the function.
Step Over	Speed up debugging if you know a particular function is bug-free, executing the current line of code in the active script or in a file that is called by the active script, without stepping into any functions called by the current line. The execution stops at the next statement. This action is available only if the execution has stopped at a breakpoint.
Finish Function	Execute the script until the current function returns. Silk Test Classic sets the focus at the line where the function returns. Try using Finish Function in combination with Step Into to step into a function and then run it.

Working with Scripts During Debugging

To run the script you are debugging	Click Debug > Run . The script runs until a breakpoint is hit, an error occurs, or it terminates.
To reset a script	Click Debug > Reset . This frees memory, frees all variables, and clears the call stack. The focus will be at the first line of the script.
To stop execution of a running script	Press Shift+Shift when running a script on the same machine or choose Debug > Abort when running a script on a different machine.

Exiting the Debugger

You can leave the debugger whenever the execution is stopped.

To exit the debugger, click **Debug > Exit** in the Silk Test Classic menu.

Breakpoints

A breakpoint is a line in the script where execution stops so that you can check the script's status. The debugger lets you stop execution on any line by setting a breakpoint. A breakpoint is denoted as a large red bullet.

One useful way to debug a script is to pause it with breakpoints, observe its behavior and check its state, then restart it. This is useful when you are not sure what lines of code are causing a problem.

During debugging, you can:

- Set breakpoints on any executable line where you want to check the call stack.
- Examine the values in one or more variables.
- See what a script has done so far.

You cannot set breakpoints on blank lines or comment lines.

Setting Breakpoints

During debugging, you can set breakpoints on most lines in the script except for blank lines or comment lines.

Setting a breakpoint on the first line of a function or test case

1. In the Silk Test Classic menu, click **Breakpoint > Add**.
2. Double-click a module name to list the functions, which are declared in the module, in the **Function** list.
3. Double-click a function name in the **Function** list to set a breakpoint on the first line of that function.

Setting a breakpoint on a line in a function or test case

Do one of the following:

- Double-click in the left margin of the line.
- Click on the line where you want to set a breakpoint, right-click, and select **Toggle Breakpoint**.
- Place the cursor on the line where you want to set a breakpoint and choose **Breakpoint > Toggle** in the Silk Test Classic menu.

Setting a breakpoint on a specific line in a script

1. In the Silk Test Classic menu, click **Breakpoint > Add**.
2. In the **Breakpoint** field, type the number of the line on which you want to set a breakpoint. For example entering 8 sets a breakpoint on the eighth line of the script.
3. Click **OK**.

Setting temporary breakpoints

Click **Debug > Run To Cursor** to set a temporary breakpoint, which is indicated by a hollow red circle in the left margin of the line where the cursor is located. When the script is executed the next time in debugging mode, the script execution stops at the marked line and the breakpoint is cleared.

Viewing Breakpoints

To view a list of all the breakpoints in a script, click **View > Breakpoints**.

Deleting Breakpoints

You can delete breakpoints in any of the following ways:

All breakpoints

1. Click **Breakpoint > Delete All**.
2. Click **Yes**.

An individual breakpoint

Place the cursor on the line where the breakpoint is set and click **Breakpoint > Toggle** .

or

Double-click in the left margin of the line

One or more breakpoints

1. Click **Breakpoint > Delete**.
2. Select one or more breakpoints from the list box and click **OK**.

Add Breakpoint Dialog Box

Use to add a breakpoint at any executable line of a function. This is available only in debugging mode.

Click **Breakpoint > Add**.

Breakpoint Type the name of a new breakpoint.

Module Displays all currently loaded scripts and include files. Click on an item from this list and a list of the file's functions and test cases display in the **Function** list.

Function Displays all functions and test cases from the module you selected at the **Module** field. Select the function that you want to enter a breakpoint into.

Delete Breakpoint Dialog Box

Use the **Delete Breakpoint** dialog box to delete one or more breakpoints. This is available only in debugging mode.

Click **Breakpoint > Delete**.

Breakpoints Lists all breakpoints from the current file by name. Select a breakpoint and then click **OK** to delete it.

Breakpoint Dialog Box

Use to view breakpoints in the current script file. You must be in debug mode to access this option.

Click **View > Breakpoints**.

Viewing Variables

To view a list of all the local variables that are in scope (accessible) from the current line, including their values, choose **View > Local Variables**.

To view a list of global variables, choose **View > Global Variables**. The variables and their values are listed in a new window.

If a variable is uninitialized, it is labelled `<unset>`.

If a variable has a complex value, like an array, Silk Test Classic might need to display its result in collapsed form. To expand or collapse the display, click **View > Expand Data** and **View > Collapse Data** in the Silk Test Classic menu or double-click the plus (+) icon.

To return to the script being debugged, press **F6** or choose **View > Module** and select the script from the list.

Changing the Value of a Variable

To change the value of an active variable, select the variable and type its new value in the **Set Value** field.

While viewing variables, you can also change their values to test various scenarios.

When you resume execution, Silk Test Classic uses the new values.

Globals Dialog Box

Use to view all global variables that are in scope (accessible) from the current source line and their current values. This is available only while Silk Test Classic is in debugging mode.

Click **View > Global Variables**.

Set Value

Type a new value for the selected variable. A script must be running for the value to be set.

List of variables

Appears in alphabetical order. If a variable is uninitialized, Silk Test Classic labels it `<unset>`. If a variable has a complex value, like an array, Silk Test Classic may display its result in collapsed form.

Click **View > Expand Data** or **View > Collapse Data** to manipulate the display or double-click on the plus/minus symbols.

Locals Dialog Box

Use the **Locals** window to view all local variables that are in scope (accessible) in the current function declaration and their current values.

Click **View > Local Variables**.

You can set a new value for the variable in the **Set Variable** text field. This is available only while Silk Test Classic is in debugging mode.

If a variable is uninitialized, it is labeled `<unset>`. If a variable has a complex value, like an array, its result is displayed in collapsed form. Use **View > Expand Data** or **View > Collapse Data** to manipulate the display, or double-click on the plus (+) and minus (-) symbols.

Expressions

If you type an identifier name, the result is the value which that variable currently has in the running script. If you type a function name, the result is the value that the function returns. Any function that you specify must return a value, and must be in scope at the current line.

Properties and methods for a class are valid in expressions, as long as the declaration for the class they belong to is included in one of the modules used by the script being debugged.

If an expression evaluates to a complex value, like an array, Silk Test Classic might display its result in collapsed form. To expand or collapse the display, click **View > Expand Data** and **View > Collapse Data** in the Silk Test Classic menu or double-click the plus (+) icon.

When a script reaches a breakpoint, you can evaluate expressions.

Evaluating Expressions

You can use the **Expressions** window to evaluate an expression during debugging and to check the result. A script must be running in order for the evaluation to occur.

1. Click **View > Expression**.

The **View** menu item is available only while Silk Test Classic is in debugging mode.

2. Type an expression into the input field in the top of the window.

- If you type an expression into the input field, the result is the value of that expression.
- If you type an variable name into the input field, the result is the value that the variable currently has in the running script.
- If you type a function name into the input field, the result is the value that the function returns. Any function that you specify must return a value, and must be in scope at the current line.

Properties and methods for a class are valid in expressions, as long as the declaration for the class they belong to is included in one of the modules used by the script being debugged.

3. Press **Enter**. The **Expression** window displays the result of the evaluation directly beneath the expression.

If an expression evaluates to a complex value, like an array, Silk Test Classic might display its result in collapsed form. To expand or collapse the display, click **View > Expand Data** and **View > Collapse Data** in the Silk Test Classic menu or double-click the plus (+) icon.

Enabling View Trace Listing

When you run a script, Silk Test Classic can record all the methods that the script invoked into a transcript. Each entry in the transcript includes the method name and the arguments passed into the method. You can use this information to debug the script, because you can see exactly which functions were actually called by the running script.

1. Click **Options > Runtime** to display the **Runtime Options** dialog box.
2. Check the **Print Agent Calls** and the **Print Tags with Agent Calls** check boxes.
3. Run the script.

The transcript contains error information and the output from print statements, and additionally lists all methods that are called by the script.

4. To check the agent trace during debugging, when execution pauses, click **View > Transcript**.

Viewing a List of Modules

1. Click **View > Module**. Silk Test Classic displays a list of modules in the **View Module** dialog. The list includes all the modules loaded at startup, which means all modules which are loaded by `startup.inc`, including `winclass.inc`, so that you can set breakpoints in functions, window class declarations, and so on.

2. Double-click the name of a module to view the module in a debug window.

View Module Dialog Box

Use the **View Module** dialog box to view the modules used by the script being debugged. This is available only while in debugging mode.

Click **View > Modules**.

Double-click on a module name to view it in a debugging window. The list includes all the modules that loaded by `startup.inc`, so you can set breakpoints in GUI functions, classes, and so forth.

Viewing the Debugging Transcripts

To see the error information in the debugging transcripts during debugging:

1. Wait until the execution is stopped.
2. In the Silk Test Classic menu, click **View > Transcript**. Silk Test Classic displays the transcript in a new window.
3. To save the contents of the transcript to a text file, choose **File > Save**.
4. To send commands to the application under test, use the **Execute** field in the **Transcript** window. Type a valid command into the field and click **Execute**.

For example, you might want to print the value of a variable or the contents of a window.

Transcript Dialog Box

Use the **Transcript** dialog box to view the debugging transcript. This dialog contains the script name and the test case names and lists error information, output from print commands, and all methods called by the script. You must be in debug mode to access this option. The contents of the transcript window are not written to disk. To save its contents to a text file, choose **File > Save**.

Click **View > Transcript**.

At the bottom of the transcript window is a text field in which you can enter any statement to execute. The results of each statement you execute appear in the transcript window.

Call Stack Dialog Box

Use to view the current call stack. This is available only in debugging mode.

Click **View > Call Stack**.

Debugging Tips

This section provides tips that might help you in debugging your tests.

Checking the Precedence of Operators

The order in which 4Test applies operators when it evaluates an expression may not be what you expect. To ensure that an expression works as expected, use parentheses or break the expression down into intermediate steps. To evaluate an expression and to check the result of the expression, click **View > Expression** in the Silk Test Classic menu.

Checking for Code that Never Executes

To check for code that never executes, step through the script with **Debug > Step Into**. For additional information, see *Stepping Into and Over Functions*.

Global and Local Variables with the Same Name

It is usually not a good programming practice to give different variables the same names. If a global and a local variable with the same name are in scope, which means accessible, at the same time, your code can access only the local variable.

To check for identical names, click **View > Local Variables** and **View > Global Variables** in the Silk Test Classic menu to see if two variables with the same name are in scope simultaneously.

Handling Global Variables with Unexpected Values

When you write a function that uses global variables, ensure that each variable has an appropriate value when the function exits. If another function uses the same variable later, and the variable has an unexpected value on entry to the function, an error could occur.

During debugging, to check that a variable has a reasonable value on entry to a function, set a breakpoint on the line that calls the function and click **View > Global Variables** in the Silk Test Classic menu to check the value of the variable.

Incorrect Usage of Break Statements

A `break` statement transfers control of the script out of the innermost nested `for`, `for each`, `while`, `switch`, or `select` statement only. `Break` exits from a single loop level, not from multiple levels. To ensure that the flow of control works as you expect, click **Debug > Step Into** in the Silk Test Classic menu to step through the script one line at a time. For additional information, see *Stepping Into and Over Functions*.

Incorrect Values for Loop Variables

When you write a `for` loop or a `while` loop, be sure that the initial, final, and step values for the variable that controls the loop are correct. Incrementing a loop variable one time more or less than you really want is a common source of errors.

To ensure that a control loop works as you expect, click **Debug > Step Into** in the Silk Test Classic menu to step through the execution of the loop one statement at a time, and watch how the value of the loop variable changes using **View > Local Variables**. See *Stepping Into and Over Functions*.

Infinite loops

To check for infinite loops, click **Debug > Step Into** in the Silk Test Classic menu to step through the script one line at a time. For additional information, see *Stepping Into and Over Functions*.

Typographical Errors

It is easy to make typographical errors that the 4Test compiler cannot catch. If a line of code does nothing, a typographical error might be the problem.

Uninitialized Variables

Silk Test Classic does not initialize variables for you. So if you have not initialized a variable on entry to a function, it will have the value `<unset>`. It is better to explicitly give a value to a variable than to trust that another function has already initialized it for you. Also, 4Test does not keep local variables after a function exits. The next time the function is called, the local variables could be uninitialized.

If you are in doubt about whether a variable has a reasonable value at a particular point, set a breakpoint there and click **View > Global Variables** or **ViewLocal Variables** in the Silk Test Classic menu to check the value of the variable.

Setting Silk Test Classic Options

This section describes the Silk Test Classic options.

Setting General Options

You can use the general options to configure aspects of the general system behavior, such as the editor and your workspace. Options that you set in the **General Options** dialog box are written to the `partner.ini` file.

1. Click **Options > General**. The **General Options** dialog box appears.
2. Check the **Create backups** check box to create a backup file each time you save a file.
The backup file has an underscore `_` appended to the extension. By default, this check box is checked.
3. Check the **Show full path** check box to see the full path of files in Silk Test Classic.
This option is useful if you have a complex directory structure, or if you work from both local and network copies. By default, this check box is checked.
4. Check the **Save files before running** check box so that all open modified files are written to disk before executing a script, suite, or test plan.
By default, this check box is checked. If you uncheck the check box, only the modified script, suite, or test plan is written to disk before running it.
5. From the **Save outline** list, select in which state outline files are saved.
Outline files include `.pln`, `.inc`, `.t`, and `.g.t` files.
6. In the **Width of tabs** field, specify the number of spaces in a tab stop.
The default is 4 spaces.
7. Check the **Show toolbar** check box to display the toolbar.
By default, this check box is checked.
8. Check the **Show toolbar tips** check box to display a on-screen description of a toolbar button (ToolTip) when you place the mouse cursor over the button.
Silk Test Classic uses the standard Windows mechanism for displaying ToolTips. You can customize the appearance of the ToolTips by using the standard **Windows Display Properties** dialog box. By default, this check box is checked.
9. In the **Project history size** field, specify the number of recent projects to display on the **File** menu.
You can specify an integer from 0 to 4. The default size is 4.
10. In the **File history size** field, specify the number of recent file actions to display on the **File** menu.
You can specify an integer from 0 to 9. The default size is 9.
11. Specify the extensions for include files in the **Include file extensions** field.
Separate the extensions with a space. Files with other extensions are treated like text files, and are displayed in the text editor only. The default include file extensions are `inc`, `lib`, and `opt`.
12. Check the **Prefer Locator** check box to use locators to resolve the window declaration, whenever both locators and tags are present.
By default, this check box is checked.
13. In the **Data file for attributes and queries** field, specify the default path for the test-plan initialization file.
Specify the full path and file name. The file extension is arbitrary. The default test-plan initialization file is `testplan.ini`.




Note: You can only specify one data file for attributes and queries at a time.

14. In the **Help files for library browser** field, specify the files that Silk Test Classic should use to compile its help file (`4test.hlp`) for the **Library Browser**. Separate the entries with a comma. The default file is `4test.txt`, which includes information for all the built-in classes and functions.
15. To disable saving object files during compilation, uncheck all check boxes in the **Auto-Complete** area as well as the **Save object files during compilation** check box on the **Runtime Options** dialog box.
 - a) Check the **Function tip** check box to display the function signature in a tooltip whenever you type an open parenthesis (after a function, test case, or method in a 4Test file.

The function signature includes the return argument type, the pass-mode, the data type, the names of any arguments, and any optional attributes that are defined. The first argument is highlighted in bold text. As you enter each argument and then type a comma, the next argument that you are expected to type is highlighted. The expected argument is always indicated with bold text; for example, if you backspace or delete an argument within your function, the expected argument is updated accordingly in the function tip.
 - b) Check the **Member list** check box to display the members from which you can select when you type a period "." after a member name in a 4Test file.

The members that you see in the **MemberList** are dependant upon the **MemberList Options** that you have selected.
 - c) Check the **Datatype list** check box to display the list of the built-in and user-defined non-winclass types whenever you type the keyword `of` after a list, an array `[]`, or a varargs; for example, `list of`.
 - d) Check the **Appstate list** check box to display a list of the currently defined application states when you type the keyword `appstate` as part of a test case declaration; for example `testcase foo () appstate`.
16. Specify how much detail appears in your **MemberList** by selecting the inheritance level from the **Inheritance Level** list.
 - Select **Below AnyWin Class** to display methods for any class derived from the `AnyWin` class. This is the default inheritance level.

 **Note:** Methods that are defined in and above the `AnyWin` class, such as `Click` and `Exist`, which are defined in the `Winclass`, are not displayed in the **MemberList**.
 - Select **All** to display the complete inheritance for members all the way up through `AnyWin` and the control classes, including the `Winclass`.
 - Select **None** to display only the members that are defined in the class of the current object and window declaration.
17. Specify which members are displayed in your **MemberList** by checking the corresponding check boxes.
 - a) Check the **Show methods** check box to display all methods that are defined in the referenced `.t`, and `.inc` files.

Methods are indicated in the **Memberlist** with a red icon.
 - b) Check the **Show window children** check box to display all window objects of type `WINDOW` that are defined in window declarations in the referenced `.t` and `.inc` files.

Works with the `parent` statement, the `with` statement, and the `this` keyword. Window children are indicated in the **Memberlist** with a yellow icon.
 - c) Check the **Show properties** check box to display all properties that are defined in the referenced `.t`, and `.inc` files.

Properties are indicated in the **Memberlist** with a blue icon.
 - d) Check the **Show variables** check box to display all variables that are defined in the referenced `.t` and `.inc` files, including native data types, data, and records.

Fields defined for records and nested records are also displayed. Variables and fields are indicated in the **Memberlist** with a red icon.

- e) Check the **Show member type** check box to display attributes for the selected members, such as the class for window children, the data type for properties and variables, and the return type for method functions.

By default, this check box is unchecked.

18. Click **OK**.

Setting the Editor Font

You can use the **Editor Font** dialog box to select a screen font in the family, size, and style of your choice.

1. Click **Options > Editor Font**. The **Editor Font** dialog box appears.
2. Select a font family from the **Font** list.
The default font is **Courier**.
3. Select a style for the selected font from the **Font style** list.
The default font style is **Regular**.
4. Select a point size for the selected font from the **Size** list.
5. Click **OK**. Silk Test Classic changes the font family, size, and style for all open windows.

While you make your selections, the **Sample** displays a line of text in the selected font family, style, and point size.

Setting the Editor Colors

You can use the **Editor Colors** dialog box to set the screen colors for various elements of 4Test code, results information, and the test plan, if available. You can select any of the provided screen colors or create your own color by modifying the RGB values of these colors.

Reset Click to revert to the default colors.

Text background Enables you to set the background color for all editor windows (.inc, .pln, .res, .t.). The default value for **Text background** is white.

1. Click **Options > Editor Colors**. The **Editor Colors** dialog box appears.
2. Select the editor item, the color of which you want to change, from the **Editor item** list.

You can change the colors of the following items:

- 4Test strings
- 4Test numbers
- 4Test keywords
- 4Test comments
- 4Test other
- Results summary
- Results output
- Results errors
- Results warnings
- Results not executed
- Results other
- Testplan test description
- Testplan statements
- Testplan comments
- Testplan other

- Difference text changed
 - Difference text added
 - Difference text deleted
 - Difference text other
 - Text background
3. Select the new color for the item.
 4. To change the RGB values of the selected color, slide the corresponding bars to the left or right, click the corresponding spin buttons, or type the value into the corresponding fields.
 5. *Optional:* Click **Reset** to revert to the default colors.
 6. Click **OK**.

Runtime Options Dialog Box


Use this dialog box to specify settings that Silk Test Classic uses when it runs a script.

Click **Options > Runtime**.

4Test area

- | | |
|----------------------|---|
| Agent name | Specifies the name of the agent on the target machine, that is, the machine on which you want to run a script or suite. Specify a value only if you want to run your script on a machine other than the host machine. |
| Network | If you are testing applications across a network, select either <code>TCP/IP</code> or <code>NetBIOS</code> as the networking protocol to use. If you choose the Open Agent as the default agent, only <code>TCP/IP</code> is available. |
| Default Agent | Specifies which agent, the Classic Agent or the Open Agent, to use by default. For example, if your test cases target an Apache Flex environment, specify the Open Agent as the default agent. For projects created prior to Silk Test Classic 2008, the Classic Agent is selected by default. You can change the agent if necessary. |
| Arguments | Specifies the arguments, if any, that you want to pass to the script at runtime. Separate multiple arguments with spaces. For example, suppose your script takes two arguments: the number of iterations to perform and the name of a test data file. In this case, you would enter <code>5 test1.dat</code> . |
| Use path | Specifies one or more paths along which Silk Test Classic searches for include files. Include files can be named in the Use files field or in a script's use statement. If you specify a path, Silk Test Classic searches the current directory and then each of the directories in the path named here. If you do not specify a search path, Silk Test Classic searches the current directory only. The syntax for a path is the same as that used by the native operating system. By default, the Use path is set to <code>c:\Program Files\Silk\<SilkTest install directory></code> . Click Browse to select an additional include path. The Use path field contains a maximum of 1024 characters. Silk Test Classic displays the first 256 characters of a directory. The 257th character is truncated. |
| Use files | Specifies the names of one or more include files for Silk Test Classic to automatically load at startup. Do not specify files in this field if you intend to include a use statement for the files in a script. Type the file names as you would type any other operating system path. You can use an absolute path or a relative path. However, it is recommended that you use a relative path. For example, to include the Java SWT extensions for the Open Agent, the relative path is <code>extend\JavaSWT\JavaSWT.inc</code> . Click Browse to select an additional include file. The Use files field contains a maximum of 1024 characters. Silk Test Classic displays the first 256 characters of a directory. The 257th character is |

truncated. When you enable extensions or configure an application, Silk Test Classic adds an include file based on the technology or browser type that you enable to the **Use files** box. For extensions that use the Open Agent, Silk Test Classic names the include file `<technology_type>.inc`. For instance, if you configure an Apache Flex application, a file named `flex.inc` is added. If you configure a Web application for an Internet Explorer browser, Silk Test Classic adds the `explorer.inc` file. Extensions that use technologies on the Classic Agent are located in the directory `C:\Users\<Current user>\Documents\Silk Test Classic Projects\<Project name>\extend\`. Technology domains that use the Open Agent are located in the directory `C:\Users\<Current user>\Documents\Silk Test Classic Projects\<Project name>\extend\<technology type>`.

Object filepath	Specifies the location from which Silk Test Classic reads and writes object files. Leave the field empty if you want to store object files in the same directory as their corresponding source files, specify an absolute path if you want to store all object files in the same directory, or specify a relative path if you want object files to be stored in a directory relative to the directory containing the source files.
GUI targets	Specifies the platforms for which you want to compile your scripts and include files (using conditional compilation). You can specify as many GUI targets as you want; separate each GUI specifier with a comma. You use this field when doing distributed testing with multiple platforms; if networking is disabled, then the field is ignored. Silk Test Classic implicitly includes the local host GUI type.
Default Browser	Specifies the Web browser to use when testing Web applications. Make sure the extension with which you are testing appears in the Default Browser field.
Save object files during compilation	<p>Creates an object file from a script or include file when it is compiled. (An object file is always created for a script or include file when it is saved.) The default is checked. Checking this option will minimize compilation time for features such as Auto-Complete and Projects, which rely on frequent compilation.</p> <p>To disable saving object files during compilation, the Auto-Complete options on the General Options dialog box as well as this option need to be unchecked.</p>
Compiler constants	Opens the Compiler Constants dialog box, where you can define constants and assign values to them. You can use the defined constants in scripts and include files anywhere you can specify an expression. Constants are evaluated and replaced with their values at compile time. To define a constant, specify its name in the Constant Name field and its value in the Value field, then click Add . You can edit or delete an existing constant by selecting it and clicking Edit or Remove .
Results area	
Directory/File	<p>Specifies the name of the directory, or the name of the directory and file that stores the results of script runs. Silk Test Classic always assigns the extension <code>.res</code> to all results files. If you supply a different extension, Silk Test Classic will override it. If you leave the field empty (the default), Silk Test Classic gives the results file the same name as the script and stores it in the same directory as the script. If you supply only a directory name, Silk Test Classic gives the results file the same name as the script and stores it in the directory you specify.</p> <p> Note: If you provide a local or remote path in this field, the path cannot be validated until script execution time. Silk Test Classic can only validate the path when the script is executing.</p>
History size	Specifies an integer representing how many sets of results to keep for a script. Once this number is reached, Silk Test Classic automatically deletes the oldest set of results each time it generates new results. A value of 0 saves all results files. Default is 5.

Write to disk after each line	If checked, writes the results file to disk whenever the script generates output, as in the case of a print statement. Selecting this option ensures that in the event of system failure, Silk Test Classic will produce a results file containing output up to the time of system failure. The disadvantage of selecting this option is that file I/O slows down script execution. By default, this check box is checked.
Find Error stops at warning	If checked, the Edit > Find Error menu option locates error messages and warnings in results files. Otherwise, the command locates error messages only. By default, this check box is checked.
Show overall summary	If checked, displays the summary of results for the entire script, suite, or test plan, including the start and elapsed time of execution, and the total number of errors and warnings. By default, this check box is checked.
Log elapsed time, thread, and machine for each output line	If checked, records this information in the results file for each line that is written. Default is unchecked. To view this information, make the results file active, then choose Results > View Options and check Elapsed time , Thread number , or Current machine .

Execution area

Minimize while running	If checked, Silk Test Classic runs minimized while you run a script, suite, or test plan. By default, this check box is unchecked.
Show detailed status window	If checked, Silk Test Classic displays the Runtime Status window while you are running a script or suite when Silk Test Classic is not minimized. By default, this check box is checked.
Save status window position	If checked, Silk Test Classic remembers the position and size of the Runtime Status window if you change it during script execution. The next time you run a script, the Runtime Status window appears in the new position and/or the new size. Default is unchecked, in which case the Runtime Status window always comes up in the same location and as the default size.
Animated Run Mode (Slow-Motion)	<p>If checked, Silk Test Classic waits one second after each interaction with the application under test is executed. Typically, you will only use this check box if you want to watch the test case run. For instance, if you want to demonstrate a test case to someone else, you might want to check this check box. You can set this check box in the Run Testcase dialog box also, which enables you to enable or disable animated run mode each time you run a test case. When you enable or disable Animated Run Mode in the Run Testcase dialog box, Silk Test Classic makes the same change in the Runtime Options dialog box. Likewise, when you enable or disable animated run mode in the Runtime Options dialog box, Silk Test Classic makes the same change in the Run Testcase dialog box. Executions of the default base state and functions that include one of the following strings are not delayed:</p> <ul style="list-style-type: none"> • BaseStateExecutionFinished • Connecting • Verify • Exists • Is • Get • Set • Print • ForceActiveXEnum • Wait

- Sleep

Debugging area

- Print agent calls** If checked, specifies whether or not you want the results file for each test run to include a list of all method calls made by your script. Each entry includes the method name and the arguments passed to it. This is a useful feature for debugging because it tells you exactly which methods were actually called by the running program. By default, this check box is unchecked.
- Print tags with agent calls** If checked and **Print agent calls** is also checked, includes tags with the method calls in your results files. By default, this check box is unchecked.

Compiler Constants Dialog Box

Use to define constants and assign values to them. You can use the defined constants in scripts and include files anywhere you can specify an expression. Constants are evaluated and replaced with their values at compile time.

Click **Options > Runtime Options** and then click **Compiler Constants**.

- Constant Name and Value** Displays the list of defined constants and the values associated with them.
- Constant Name** Type the name of the constant you want to add to the list.
- Value** Type the value of the constant you specified in the **Constant Name** box.
- Edit** Click to modify the constant you selected from the **Constant Name** and **Value** box. You must select a constant from the list before the **Edit** button is available.
- Remove** Click to delete the constant you selected from the **Constant Name** and **Value** box. You must select a constant from the list before the **Remove** button is available.
- Add** Specify a **Constant Name** and **Value** in the appropriate boxes and then click **Add** to add the new constant to the list.

Agent Options Dialog Box

Use to set global options for how the agent interacts with the application under test. Options you specify within the tabs are saved in the `partner.ini` file. If you work on multiple projects, you might want to have a custom set of agent options for each project. To do this, save your options in a file with a name other than `partner.ini`.

Click **Options > Agent**.

Any option you set in the **Agent Options** dialog box can be changed by the same option set to a different value within a script.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the `Window timeout` to 5 seconds on the **Timing** tab the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT, 5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option, and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click **Copy to Clipboard** to copy the statement to the clipboard.

Timing Tab

The **Timing** tab is available on the **Agent Options** dialog box and contains the following options:

- Window timeout (seconds)** Specifies the number of seconds Silk Test Classic waits for a window to appear and be in the correct state. If a window does not appear in the correct state within the specified timeout, Silk Test Classic raises an exception. The correct state of the window depends on how you set the options on the **Verification** tab, which determine whether Silk Test Classic checks whether a window is enabled, active, exposed, or unique. Default is 5 seconds, unless you have enabled enhanced support for Visual Basic, in which case the default is 20 seconds for 32-bit applications.
- Window retry interval (seconds)** Specifies the number of seconds Silk Test Classic waits between attempts to verify a window, if the window does not exist or is in the incorrect state. Silk Test Classic continues trying to find the window until the time specified by the **Window timeout** option is reached. The correct state of the window depends on how you set the options on the **Verification** tab, which determine whether or not Silk Test Classic checks whether a window is enabled, active, exposed, or unique. Default is 0.06 seconds.
- Keyboard event delay (seconds)** Specifies the delay used before each keystroke in a script. Default is 0.02 seconds. You can specify a number in increments of .001 from .001 to 999.999 seconds, inclusive.
- Setting a keyboard event delay is necessary only if objects seem not to be recognized. This often appears as a `win.SetActive()` being executed before the action to display the window or the wrong window may be made active. The amount you may need to adjust the timing depends upon the speed of the machine you are working on and the application that you are testing. We recommend starting with a delay of .001 and incrementing only by .001 if necessary.
- Setting this value to .001 is necessary only for client/server application testing – if you are testing a Web application, a setting of .001 radically slows down the browser. However, setting this to 0 (zero) may cause basic application testing to fail.
- Mouse event delay (seconds)** Specifies the delay used after each mouse event (when the mouse is moved to an object) in a script. The delay affects moving the mouse, pressing buttons, and releasing buttons. Default is 0.02 seconds.
- Application ready timeout (seconds)** Specifies the number of seconds that the agent waits for an application to become ready. If the application is not ready within the specified timeout, Silk Test Classic raises an exception. Default is 10 seconds. This option applies only if the application or extension knows how to communicate to the agent that it is ready. To find out whether the extension has this capability, see the documentation that comes with the extension. To require the agent to confirm the ready state of an application, select the **Verify That an Application is Ready (Requires an Extension)** check box on the **Verification** tab.
- Application ready retry interval (seconds)** Specifies the number of seconds Silk Test Classic waits between attempts to verify that an application is ready. Silk Test Classic continues to test the application for readiness if it is not ready until the time specified by the **Application ready timeout** option is reached. Default is 0.1 seconds.

Timeout to wait for active window (seconds)	This functionality is supported only if you are using the Classic Agent. Enter the maximum time, in seconds, to wait for a window to become active.
Retry interval to wait for active window (seconds)	This functionality is supported only if you are using the Classic Agent. Enter the maximum time, in seconds, to wait for a window to become active before trying to verify the object again.
Timeout to wait for an enabled window (seconds)	Enter the maximum time, in seconds, to wait for a window to become enabled.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT, 5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard Copies the 4Test statement which sets the option to the clipboard.

Verification Tab

The **Verification** tab is available on the **Agent Options** dialog box and contains the following options:

Verify that windows are active	If checked, verifies that windows are active before interacting with them. Default is checked.
Verify that windows are enabled	If checked, verifies that windows are enabled before interacting with them. Default is checked.
Verify that windows are exposed	If checked, verifies that windows are exposed (that is, not covered, obscured, or logically hidden by another window) before interacting with them. Default is checked.
Verify that a tag uniquely identifies a window	If checked, verifies that each tag used by the script matches only one window. Default is checked.
Verify that coordinates passed to a method are inside the window	If checked, verifies that the coordinates, passed to a method, are inside the window before the mouse is pressed. If checked and coordinates fall outside the window, Silk Test Classic raises the <code>E_COORD_OUTSIDE_WINDOW</code> exception. Typically, you use the checking feature unless you need to be able to pass coordinates outside of the window, such as negative coordinates.



Note: The `MoveMouse`, `PressMouse`, and `ReleaseMouse` methods never verify their coordinates. Default is checked.

Verify the class tag for methods of class Control If checked, verifies that objects are of the specified type before interacting with them. This option is unchecked and disabled.

Verify that an application is ready (requires an extension) If checked, synchronizes the agent with the application under test. Calls made to the agent will not proceed until the application is ready. This option applies only if you have an extension enabled in the **Extensions** dialog box. Default is checked.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT,5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard Copies the 4Test statement which sets the option to the clipboard.

Close Tab

The **Close** tab is available from the **Agent Options** dialog box and contains the following options:

List of buttons used to close a window Specifies a list of strings representing the list of buttons used to close windows closed with the `Close`, `CloseWindows`, and `Exit` methods. The preferred way to specify these buttons is with the `IsCloseWindowButtons` variable in the object's declaration. Default is `Cancel, Close, Exit, Done`. If the UI of the application is localized, you need to replace the strings with the localized strings.

Keystrokes used to close a dialog box window Specifies a list of strings representing the keystroke sequence used to close dialog boxes (used by `Close`, `CloseWindows`, and `Exit`). The preferred way to specify this keystroke sequence is with the `IsCloseDialogKeys` variable in the object's declaration. Default is `<Esc>`.

List of menus used to close a window Specifies a list of strings representing the list of menu items used to close windows with `Close`, `CloseWindows`, and `Exit`. The preferred way to specify these menu items is with the `IsCloseWindowMenus` variable in the object's declaration. Default is `File/Exit*, File/Quit*`.

List of buttons used to close a confirmation window Specifies a list of strings representing the list of buttons used to close confirmation dialog boxes, which means dialog boxes or message boxes that appear when closing windows with the `Close`, `CloseWindows`, and `Exit` methods. The preferred way to specify these buttons is with the `IsCloseConfirmButtons` variable in the object's declaration. Default is `No`.

Name of Close item on system menu Specifies a list of strings representing the list of menu items on the system menu used to close windows with the `Close`, `CloseWindows`, and `Exit` methods. Default is `Close`.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT, 5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard Copies the 4Test statement which sets the option to the clipboard.

Bitmap Tab

The **Bitmap** tab is available from the **Agent Options** dialog box and contains the following options:

Bitmap match count	Specifies an integer representing the number of consecutive snapshots that must be the same for the bitmap to be considered stable. Snapshots are taken up to the number of seconds specified by the <code>Bitmap match timeout</code> option, with a pause specified by <code>Bitmap match interval</code> occurring between each snapshot. This option affects the <code>CaptureBitmap</code> , <code>GetBitmapCRC</code> , <code>VerifyBitmap</code> , and <code>WaitBitmap</code> methods. Default is 5.
Bitmap match interval (seconds)	Specifies a number representing the time interval between snapshots to use for ensuring the stability of the bitmap image. The snapshots are taken up to the time specified by the <code>Bitmap match timeout</code> option. This option affects the <code>CaptureBitmap</code> , <code>GetBitmapCRC</code> , <code>VerifyBitmap</code> , and <code>WaitBitmap</code> methods. Default is 0.1 seconds.
Bitmap match timeout (seconds)	Specifies a number representing the total time allowed for a bitmap image to become stable. During the time period, Silk Test Classic takes multiple snapshots of the image, waiting the number of seconds specified by the <code>Bitmap match interval</code> option between snapshots. If the timeout period is reached before the number of bitmaps specified by the <code>match count</code> option, Silk Test Classic stops taking snapshots and raises the exception <code>E_BITMAP_NOT_STABLE</code> . This option affects the <code>CaptureBitmap</code> , <code>GetBitmapCRC</code> , <code>VerifyBitmap</code> , and <code>WaitBitmap</code> methods. Default is 5 seconds.
Bitmap compare tolerance (pixels)	Specifies an integer representing the number of pixels of difference below which two bitmaps are considered to match. If the number of pixels that are different is smaller than the number specified with this option, the bitmaps are considered identical. The maximum tolerance is 32,767 pixels. The <code>bitmap pixel tolerance</code> is used by the <code>VerifyBitmap</code> and <code>WaitBitmap</code> methods, as well as the <code>SYS_CompareBitmap</code> function. Default is 0 pixels.

You can also get and set bitmap options using the `GetOption` and `SetOption` methods.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT, 5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard Copies the 4Test statement which sets the option to the clipboard.

Bitmap Agent Options

The following table lists the agent options which determine the behavior of bitmap verification.

Option	Description
opt_bitmap_match_count	The number of successive snapshots that must be the same for the bitmap to be considered stable.
opt_bitmap_match_interval	The time interval between snapshots to use for ensuring the stability of the image.
opt_bitmap_match_timeout	The total time allowed for a bitmap to become stable.
opt:bitmap_pixel_tolerance	The number of pixels that is allowed to be different for two bitmaps to be considered equivalent.

To set these options globally for a group of scripts, use the **Agent Options** dialog box. To set these options locally within a script or test case, use the `SetOption` method. For example:

```
Agent.SetOption (OPT_BITMAP_MATCH_COUNT, 3)
```

Synchronization Tab

This functionality is supported only if you are using the Open Agent.

The **Synchronization** tab is available on the **Agent Options** dialog box contains the following options:

Synchronization mode Select the synchronization algorithm for the ready state of a Web application. The synchronization algorithm configures the waiting period for the ready state of an invoke call.

HTML mode Using the HTML mode ensures that all HTML documents are in an interactive state. With this mode, you can test simple web pages. If more complex scenarios with Java script are used, it might be necessary to manually script synchronization functions.

AJAX mode Using the AJAX mode eliminates the need to manually script synchronization functions (such as wait for objects to appear or disappear, wait for a specific property value, and so on), which eases the script creation process dramatically. This automatic synchronization is also the base for a successful record and replay approach without manual script adoptions.

Synchronization timeout (seconds) Enter the maximum time, in seconds, to wait for an object to be ready.

Synchronization exclude list Type the entire URL or a fragment of the URL for any service or Web page that you want to exclude.

Some AJAX frameworks or browser applications use special HTTP requests, which are permanently open in order to retrieve asynchronous data from the server. These requests may let the synchronization hang until the specified synchronization timeout expires. To prevent this situation, either use the HTML synchronization

mode or specify the URL of the problematic request in the Synchronization exclude list setting.

For example, if your web application uses a widget that displays the server time by polling data from the client, permanent traffic is sent to the server for this widget. To exclude this service from the synchronization, determine what the service URL is and enter it in the exclusion list.

For example, you might type:

```
http://example.com/syncsample/timeService
timeService
UICallBackServiceHandler
```

Separate multiple entries with a comma.

If your application uses only one service, and you want to disable that service for testing, you must use the HTML synchronization mode rather than adding the service URL to the exclusion list.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT,5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard Copies the 4Test statement which sets the option to the clipboard.

Advanced Tab

This functionality is supported only if you are using the Open Agent.

The **Advanced** tab is available on the **Agent Options** dialog box and contains the following options:

Use Microsoft Accessibility Check to turn on Microsoft Accessibility control recognition for a Win32 application while the application is running.

Remove focus on capture text Check to remove the focus from a window before text is captured.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT,5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard

Copies the 4Test statement which sets the option to the clipboard.

Other Tab

This functionality is supported only if you are using the Classic Agent.

The **Other** tab is available on the **Agent Options** dialog box and contains the following options:

Tolerance to use when sizing windows (pixels)

Specifies an integer representing the number of pixels allowed for a tolerance when a resized window does not end at the specified size. For some windows and GUIs, you can't always resize the window to the particular size specified. If the ending size is not exactly what was specified and the difference between the expected and actual sizes is greater than the tolerance, Silk Test Classic raises an exception. Windows cannot be sized smaller than will fit comfortably with the title bar. Default is 0 pixels.

Tolerance to use when moving windows (pixels)

Specifies an integer representing the number of pixels allowed for a tolerance when a moved window does not end up at the specified position. For some windows and GUIs, you can't always move the window to the particular pixel specified. If the ending position is not exactly what was specified and the difference between the expected and actual positions is greater than the tolerance, Silk Test Classic raises an exception.

The tolerance can be set through the **Control Panel**, by setting the desktop window granularity option. If the granularity is zero, you can place a window at any pixel location. If the granularity is greater than zero, the desktop is split into a grid of the specified pixels in width, determining where a window can be placed. In general, the tolerance should be greater than or equal to the granularity. Default is 0 pixels.

Events used to invoke popup menus

Specifies a string, which is the command (keystrokes or mouse buttons) used to display pop-up menus. To use mouse buttons, specify <button1>, <button2>, or <button3> in the command sequence. Default is <Button2><Up><Down>. If you are testing Java applications or applets, we recommend that you set this options to <Button2>.

Pick menu before getting menu item information

If checked, picks the menu before checking whether an item on it exists, is enabled, or is checked. You might see menus pop up on the screen even though your script does not explicitly call the `Pick` method. Default is unchecked.

Pick dropdowns before getting item information

If checked, drops down the combo box before trying to access the contents. This is usually not needed, but some combo boxes only get populated after they are dropped down. Check this option if you are having problems getting the contents of a combo box, such as the **Location** combo box in Internet Explorer. Default is unchecked.

Consider case when matching items in controls

Check this option to consider case when looking for an item in a control. Default is unchecked.

Show windows which are out of view

If checked, allows controls that are not currently scrolled into view to be seen by the agent. This option is useful for testing Web applications. If unchecked, controls not currently in view are invisible. Default is checked.

Automatically scroll windows into view

If checked, scrolls a control into view before recording events against it or capturing its bitmap. This option is useful for testing web applications. This option applies only if **Show windows which are out of view** is enabled. This option is useful for testing web applications in which a dialog contains a scroll bar. Default is unchecked.

Enable communication with SilkBean If checked, enables communication with the SilkBean for all scripts in a test session. The SilkBean is a utility that allows you to perform cross-platform testing of 100% pure Java controls in standalone Java applications. This option applies only if you enable Java support on Silk Test Classic. Default is unchecked.

We strongly recommend that you keep the default settings for **Show windows which are out of view** and **Automatically scroll windows into view**. If you do change the values, save them in a custom options set.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT,5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard Copies the 4Test statement which sets the option to the clipboard.

Compatibility Tab

This functionality is supported only if you are using the Classic Agent.

The **Compatibility** tab is available from the **Agent Options** dialog box. Use the following options only if you are upgrading from a previous release of Silk Test Classic and want to use functionality that matches the previous release.

Don't group radio buttons into a list If unchecked, treats radio buttons as a group of class `RadioList`. Otherwise, uses the `Release 1.x` method of treating radio buttons as individual objects. Default is unchecked.

Use release 1.x version of the class library If checked, uses QA Partner Release 1 versions of `GetChildren`, `GetClass`, and `GetParent`. Otherwise, uses Release 2 and later versions. Default is unchecked.

Use release 1.x window tags If checked, generates and operates on tags compatible with Silk Test Classic releases earlier than QA Partner Release 2. Otherwise, uses Release 2 or later tags. Default is unchecked. The Release 2 algorithm affects tags that use index numbers and some tags that use captions. In general, Release 2 tags are more portable, while the earlier algorithm generates more platform-dependent tags. Use this option only if you must run old scripts without any changes.

Don't trim spaces when getting items in controls If checked, leading and trailing spaces are not trimmed from items on windows. Default is unchecked.

Add 'Window tags' to the Record menu If checked, includes the **Record Window Tags** menu item on the **Record** menu. Selecting the **Record Window Tags** menu item opens the **Record Window Tags** dialog. Use the **Record Window Tags** dialog to capture and paste window tags for individual objects into your script or test frame. This dialog is equivalent to the QA Partner Release 1.x **Paste Window** dialog box, allowing you to paste Release 1.x style window tags into your script. Default is unchecked.

Only use Agent-based clicks (not API-based clicks)

If checked, use API-based clicks when available. An API click is generated internally by the browser, instead of by the agent. API clicks are more reliable than agent clicks, which can click the wrong location of an object. API-based clicks are for browsers only. If cleared, use agent-based clicks even when API-based clicks are available. Default is unchecked.

Use ANSI call

If checked, converts all string parameters (return type, string arguments, strings in record structures that are used as arguments) from W (wide-character, Unicode) to A (ANSI) character format before calling the DLL function internally. Return values and `pass-mode` variables set to `inout` that are strings or contain strings within record structures get converted back from A (ANSI) format to W (wide-character) format after calling the DLL. Using this check box makes an ANSI call transparent, as the conversion is done internally. Default is unchecked.

The DLL calling stack does not support return values that are pointers to a record structure. To determine if the function succeeded, use the `inout` or `out pass-mode` variable and use the `return-type` variable as a flag.



Important: If you use DLL functions where you specify string sizes in bytes, remember that W (wide-character) strings are assigned 2 bytes per character, and A (ANSI) strings are assigned 1 byte per character. If you pass a buffer of 1000 characters via an ANSI call, you use 1000 bytes of space, not 2000 bytes. But if you return a string of 1000 characters from a DLL, the W (wide-character) buffer will have 2000 bytes.

4Test statement which sets the selected option

When you set an option, this field displays the 4Test statement that represents the selected option. For example, if you set the **Window timeout** to 5 seconds on the **Timing** tab, the following 4Test statement displays:

```
Agent.SetOption(OPT_WINDOW_TIMEOUT,5)
```

The statement uses the `SetOption` method to operate on the agent object. `OPT_WINDOW_TIMEOUT` is the internal 4Test name of the `Window Timeout` option and 5 is the current, default value of the option, five seconds.

This code is useful if you want to set a local value for a given agent. You can type or change the code yourself, then click the **Copy to Clipboard** button to copy the statement to the clipboard.

Copy to Clipboard

Copies the 4Test statement which sets the option to the clipboard.

Extensions Dialog Box

This functionality is supported only if you are using the Classic Agent.

Use to enable extensions for applications under test.

Click **Options > Extensions**.

The **Extensions** dialog box displays the runtime environments for all extensions that you have installed, or that work with the Classic Agent. The environments listed in the dialog box are used for running the applications that you want to test. The extensions enable Silk Test Classic to recognize the non-standard controls in your environment. If you are using a project, the information you specify in the **Extensions** dialog box is stored in the `partner.ini` file. If you are not using a project, the information you specify in the **Extensions** dialog box is stored in the `extend.ini` file.

Application Column

Displays the browser and runtime environments that the installation program installed and recognizes automatically. For applications and runtime environments that do not use standard names, or for stand-alone applications, you must use the **New** button to manually

add the name of the executable or DLL file to the **Extensions** dialog box. If the executable name contains spaces, you must enclose the name in quotation marks.

Primary Extension Column

For each application listed in the **Application** column, the **Primary Extension** column displays the currently selected primary extension, or `Disabled` or `None`, if applicable. To change a primary extension, click in the **Primary Extension** column, then use the embedded list box to select one of the available options.

Keep in mind that you cannot enable more than one primary extension for an application. Primary extensions are the ones listed in the list boxes of the **Extensions** dialog box, such as `DOM` and `.NET`. Secondary extensions, which are checked with the check boxes in the **Extensions** dialog box, `Java`, `ActiveX`, and `Accessibility`, are the only ones that can co-exist with another extension. Thus, for example, an embedded browser in a `.NET` application is not allowed, since both `IE DOM` and `.NET` are primary extensions.

Also, remember that not all of the **Primary Extensions** in the list box can be applied to a new application. Only `IE DOM`, which has superseded the `IE4 OCX` extension for embedded browser applications, or `.NET` extensions make sense for new applications.

Other Extensions

You can use the check boxes in the **Other Extensions** columns to enable other extensions required for testing each application that is launched in the runtime environment listed in the **Application** column. When a check box in the **Other Extensions** column is checked, the corresponding button in the **Options** area is enabled.

Java Enables Java support for an application running on the host machine or an applet running in a browser. When the **Java** check box is checked, the **Java** button in the **Options** area is enabled. Click the **Java** button to display the **Extension Options** dialog box for Java and select Java extension options for the currently selected application.

ActiveX Enables ActiveX and Visual Basic support for an application running on the host machine. When the **ActiveX** check box is checked, the **ActiveX** button in the **Options** area is enabled. Click the **ActiveX** button to display the **Extension Options** dialog box for ActiveX and select ActiveX extension options for the currently selected application.

Accessibility Enables Windows Accessibility. When the **Accessibility** check box is checked, the **Accessibility** button in the **Options** area is enabled. Click the **Accessibility** button to display the **Windows Accessibility** dialog box.

.NET Enables `.NET` support for an application running on the host machine.

Options area

The buttons in the **Options** area are disabled until a corresponding extension is enabled for the currently selected application.

Extension Enabled only when a browser is enabled in the **Primary Extension** column for the currently selected application. Click **Extension** to display the **DOM Extensions** dialog box and specify options for the `DOM` extension.

Java Enabled only when a **Java** check box is checked in the **Other Extensions** column for the currently selected application. Click **Java** to display the **Extension Options** dialog box for Java and redirect output from the Java console to a local file. This allows you to scroll and copy text more easily.

ActiveX Enabled only when an **ActiveX** check box is checked in the **Other Extensions** column for the currently selected application. Click **ActiveX**

to display the **Extension Options** dialog box for ActiveX and select ActiveX and Visual Basic options.

Accessibility	Enabled only when an Accessibility check box is checked in the Other Extensions column for the currently selected application. Click Accessibility to display the Windows Accessibility dialog box.
New	Displays the Extension application dialog box where you can add one or more runtime environments to the Extensions dialog box for testing applications.
Duplicate	Active only when you select a runtime environment in the Application column that you entered manually. You cannot duplicate runtime environments that are installed by default. Click Duplicate to add a new application that has all the same settings, primary and other extensions and other options, as the selected application, but with a different name.
Remove	Active only when you select a runtime environment in the Application column that you entered manually. You cannot remove runtime environments that are installed by default. Click Remove to delete a runtime environment from the dialog box.
Details	Active only when you select a runtime environment with an enabled Primary Extension. Click Details to display detailed information about the runtime environment, including version, help text, and executable modules.

Extension Details Dialog Box

Use the **Extension Details** dialog box to view more information about the selected extension. Access this dialog box by clicking **Options > Extensions**, enabling and selecting the extension for which you want to view details, and then clicking **Details**.

This dialog box contains:

File	Displays the file name associated with the selected extension.
Version	Displays the version of the file associated with the selected extension.
Include	Displays the name of the include file associated with the selected extension, if applicable.
Help text	Displays the file name of the help text associated with the selected extension, if applicable.
Prop set	Displays the name of the property set associated with the selected extension.
Modules	Displays the name of modules associated with the selected extension, if applicable.

Setting Recording Options for the Open Agent

This functionality is supported only if you are using the Open Agent.

You can set the recording options to optimize recording with the Open Agent in the following two ways:

- In the **Recording Options** dialog box.
- Within a script, by using the `SetOption` method.

Using `SetOption` overrides the value specified for the option in the **Recording Options** dialog box. If you do not set an option with `SetOption`, the value specified in the **Recording Options** dialog box is the default.

To set the recording options in the **Recording Options** dialog box, perform the following actions:

1. Click **Options > Recorder**. The **Recording Options** dialog box appears.
2. To set **Ctrl+Shift** as the shortcut key combination to use to pause recording, check the **OPT_ALTERNATE_RECORD_BREAK** check box.

By default, **Ctrl+Alt** is the shortcut key combination.



Note: For SAP applications, you must set **Ctrl+Shift** as the shortcut key combination.

3. To record absolute values for scroll events, check the **OPT_RECORD_SCROLLBAR_ABSOLUT** check box.
4. To record mouse move actions for web applications, Win32 applications, and Windows Forms applications, check the **OPT_RECORD_MOUSEMOVES** check box. You cannot record mouse move actions for child technology domains of the xBrowser technology domain, for example Apache Flex and Swing.
5. If you record mouse move actions, in the **OPT_RECORD_MOUSEMOVE_DELAY** text box, specify how many milliseconds the mouse has to be motionless before a `MouseMove` is recorded.

By default this value is set to 200.

6. To record text clicks instead of clicks, check the **OPT_RECORD_TEXT_CLICK** check box.
Recording text clicks is enabled by default, and is the fallback mechanism when testing applications which display texts. Micro Focus recommends to leave this check box checked.
7. To resize the application under test (AUT) when a recording session starts, check the **OPT_RESIZE_APPLICATION_BEFORE_RECORDING** check box.

This check box is checked by default, enabling the recording sidebar to display next to the AUT. When this check box is unchecked, the AUT and the recording sidebar might overlap.

8. Define custom attributes for recording.

- a) Select the **Custom Attributes** tab.
- b) Select the technology domain of the application that you are testing.
For example, to set custom attributes for a web application, select **xBrowser**.
- c) Add the attributes that you want to use to the list.

Separate attribute names with a comma.

Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index might change when another object is added. If custom attributes are available, the locator generator uses these attributes before any other attribute. The order of the list also represents the priority in which the attributes are used by the locator generator. If the attributes that you specify are not available for the objects that you select, Silk Test Classic uses the default attributes for the application that you are testing.



Note: You cannot set custom attributes for Apache Flex applications.

9. Set the classes that you want to ignore during recording and replay.

- a) Select the **Transparent Classes** tab.
- b) Add the names of any classes that you want to ignore to the list.

Separate class names with a comma.

10. Specify recording options for web applications.

- a) Select the **Browser** tab.
- b) Add names of attributes that you want to ignore during recording to the **Locator attribute name exclude list**.

For example, if you do not want to record attributes named *height*, add *height* to the list. Separate attribute names with a comma.

- c) Add values of attributes that you want to ignore during recording to the **Locator attribute value exclude list**.

- For example, if you do not want to record attributes that have the value *x-auto*, add *x-auto* to the list.
- d) Check the **OPT_XBROWSER_LOWLEVEL** check box to record native user input instead of DOM functions.

For example to record `Click` instead of `DomClick` and `TypeKeys` instead of `SetText`.

If your application uses a plug-in or AJAX, use native user input. If your application does not use a plug-in or AJAX, Micro Focus recommends using high-level DOM functions, which do not require the browser to be focused or active during playback. As a result, tests that use DOM functions are faster and more reliable.

- e) Specify the maximum length for locator attribute values in the **OPT_XBROWSER_LOCATOR_MAX_ATTRIBUTE_VALUE_LENGTH** field.
- f) Check the **OPT_XBROWSER_ENABLE_SMART_CLICK_POSITION** check box to automatically search for a free click spot on the object.

If disabled, the click is always made in the center of the object, and might possibly be performed on an object layered over the target.

11. Specify recording options for WPF applications.

- a) Check the **OPT_WPF_PREFILL_ITEMS** check box to pre-fill items in a `WPFItemsControl`, for example a `WPFComboBox` or `WPFListBox`, during recording and playback.
- b) Add the names of custom classes that you want to expose during recording and playback to the **Custom WPF class names** list.

12. Click OK.

Setting Recording Options for the Classic Agent

This functionality is supported only if you are using the Classic Agent.

You can set the recording options to optimize recording with the Classic Agent in the following two ways:

- In the **Recorder Options** dialog box.
- Within a script, by using the `SetOption` method.

Using `SetOption` overrides the value specified for the option in the **Recorder Options** dialog box. If you do not set an option with `SetOption`, the value specified in the **Recorder Options** dialog box is the default.

To set the recording options in the **Recorder Options** dialog box, perform the following actions:

1. Click **Options > Recorder**. The **Recorder Options** dialog box opens.
2. To set **Ctrl+Shift** as the shortcut key combination to use to pause recording, check the **Change hotkey to Ctrl+Shift** check box.

By default, **Ctrl+Alt** is the shortcut key combination.

3. To record the tags that are specified in the **Record Window Declarations Options** dialog box, check the **Record multiple tags** check box.

If checked, Silk Test Classic records multiple tags whenever recording. If this check box is disabled, see the description of the multiple tags agent option.



Note: If you change the setting for **Record Multiple Tags** here, the check box on the **Record Window Declarations Options** dialog box is automatically updated.

4. To add new declarations to the INC file during recording, check the **Auto Declaration** check box.
5. To verify the test application using properties instead of attributes, check the **Verify using properties** check box.

This option is checked automatically if you have enabled enhanced support for Visual Basic. This feature requires properties for verification. You cannot uncheck the **Verify using properties** check box without disabling enhanced support for Visual Basic.

6. Specify the file that contains the definitions for the used property sets in the **Data file for property sets** field.
7. To record events at a lower level for selected controls, check the corresponding check boxes in the **Recorded Events** list.
For example, you might want to record a click in a check box, instead of recording an actual selection. If you specify that you want to record only low-level events in check boxes, Silk Test Classic records something like the following when you select a check box: `Find.CaseSensitive.Click (1, 41, 10)`. If you are using a high-level event, Silk Test Classic records something like the following:
`Find.CaseSensitive.Check ()`.
8. Uncheck the **Ignore mouse move events** check box to record mouse movements.
If this check box is unchecked, Silk Test Classic records mouse movements that cannot be built into higher-level actions and that occur while a mouse button is pressed. Leave this check box checked unless you are testing an application, such as a drawing application, where mouse movements themselves are significant.
9. To record mouse movements that cannot be built into higher-level actions and that occur while a mouse button is pressed when you select the **Record Testcase** and **Record Actions** commands, uncheck the **Ignore mouse move events** check box.
Leave the check box checked unless you are testing an application where mouse movements themselves are significant.
10. Uncheck the **Don't record BeginDrag/EndDrag** check box to record `BeginDrag` and `EndDrag` methods when you press a mouse button on an object and do a drag operation on a listview, treeview, or list box.
11. Click **OK**.

Setting Replay Options for the Open Agent

This functionality is supported only if you are using the Open Agent.

You can set the replay options to optimize replaying tests with the Open Agent.

1. Click **Options > Agent**. The **Agent Options** dialog box appears.
2. Click the **Replay** tab.
3. Select a **Replay mode** from the list.
 - **Default**: Use this mode for the most reliable results. Each control uses the best method for the control type, by using either the mouse and keyboard (low level) mode or the API (high level) mode.
 - **High level**: Use this mode to replay each control using the API.
 - **Low level**: Use this mode to replay each control using the mouse and keyboard.
4. To ensure that the window is active before a call is executed, check the **Ensure window is active** check box.
5. Click **OK**.

Defining which Custom Locator Attributes to Use for Recognition

The Open Agent includes a sophisticated locator generator mechanism that guarantees locators are unique at the time of recording and are easy to maintain. Depending on your application and the frameworks that you use, you might want to modify the default settings to achieve the best results and stable recognition of the controls in your application. You can use any property that is available in the respective technology as a custom attribute, given that the property is either a number, like an integer or a double, a string, an item identifier, or an enumeration value.

A well-defined locator relies on attributes that change infrequently and therefore requires less maintenance. Using a custom attribute is more reliable than other attributes like caption or index, since a caption will change when you translate the application into another language, and the index might change when another object is added.

In xBrowser, WPF, Java SWT, and Swing applications, you can also retrieve arbitrary properties, such as a `WPFButton` that defines `myCustomProperty`, and then use those properties as custom attributes. To achieve optimal results, the application developers can add a custom automation ID to the controls that you want to interact with in your test. In Web applications, the application developers can add an attribute to controls that you want to interact with, such as `<div myAutomationId="my unique element name" />`. This approach can eliminate the maintenance associated with locator changes. Or, in Java SWT, the UI developer can define a custom attribute, for example `testAutomationId`, for a widget that uniquely identifies the widget in the application. You can then add that attribute to the list of custom attributes, in this case `testAutomationId`, and you can then identify controls by that unique ID. This approach can eliminate the maintenance associated with locator changes.

If more than one objects have the same custom attribute value assigned, all the objects with that value will be returned when you call the custom attribute. For example, if you assign the unique ID `loginName` to two different text boxes, both text boxes will be returned when you call the `loginName` attribute.

To define which custom attributes of a locator should be used for the recognition of the controls in your AUT:

1. Click **Options > Recorder** and then click the **Custom Attributes** tab.
2. From the **Select a tech domain** list box, select the technology domain for the application that you are testing.



Note: You cannot set custom attributes for Flex or Windows API-based client/server (Win32) applications.

3. Add the attributes that you want to use to the list.

If custom attributes are available, the locator generator uses these attributes before any other attribute. The order of the list also represents the priority in which the attributes are used by the locator generator. If the attributes that you specify are not available for the objects that you select, Silk Test Classic uses the default attributes for the application that you are testing. Separate attribute names with a comma.

4. Click **OK**. You can now record or manually create a test case.

Setting Classes to Ignore

To specify the names of any classes that you want to ignore during recording and replay:

1. Click **Options > Recorder**. The **Recording Options** dialog box opens.
2. Click the **Transparent Classes** tab.
3. In the **Transparent classes** grid, type the name of the class that you want to ignore during recording and replay.
Separate class names with a comma.
4. Click **OK**.

Custom Controls Dialog Box

This functionality is supported only if you are using the Open Agent.

Options > Manage Custom Controls.

Silk Test Classic supports managing custom controls over the UI for the following technology domains:

- Win32
- Windows Presentation Foundation (WPF)
- Windows Forms
- Java AWT/Swing
- Java SWT

In the **Frame file for custom class declarations**, define the frame file into which the new custom classes should be generated.

When you map a custom control class to a standard Silk Test class, you can use the functionality supported for the standard Silk Test class in your test. The following **Custom Controls** options are available:

Option	Description
Silk Test base class	Select an existing base class to use that your class will derive from. This class should be the closest match to your type of custom control.
Silk Test class	Enter the name to use to refer to the class. This is what will be seen in locators.
Custom control class name	Enter the fully qualified class name of the class that is being mapped. You can use the wildcards ? and * in the class name.
Use class declaration	This option is available only for Win32 applications. By default <code>False</code> , which means the name of the custom control class is mapped to the name of the standard Silk Test class. Set this setting to <code>True</code> to additionally use the class declaration of the custom control class.



Note: After you add a valid class, it will become available in the **Silk Test base class** list. You can then reuse it as a base class.

Example: Setting the options for the UltraGrid Infragistics control

To support the UltraGrid Infragistics control, use the following values:

Option	Value
Silk Test base class	Control
Silk Test class	UltraGrid
Custom control class name	Infragistics.Win.UltraWi nGrid.UltraGrid

Property Sets Dialog Box

This functionality is supported only if you are using the Classic Agent.

Use the **Property Sets** dialog box to view existing property sets and create, modify, combine, and delete property sets. Property sets are used to verify properties in test cases.

To open the **Property Sets** dialog box, click **Options > Property Sets**.

The **Property Sets** dialog box includes the following items:

Item	Description
Property Sets	Displays the list of available property sets.

Item	Description
New	Click to access the New Property Set dialog box where you can create new property sets.
Combine	Click to access the Combine Property Sets dialog box where you can combine existing property sets into a new property set.
Edit	Click to access the Edit Property Set dialog box where you can modify and delete property sets.
Remove	Click to delete the selected property set. Select the property set that you want to remove, click Remove , and then click Yes to delete the selected property set.

New Property Set Dialog Box

This functionality is supported only if you are using the Classic Agent.

Use the **New Property Set** dialog box to create property sets, which are used to verify properties in test cases. Use the **Edit Property Set** dialog box to modify and delete property sets.

To access the **New Property Set** dialog box, click **Options > Property Sets > New** or click **Define** on the **Verify Window** dialog box.

To access the **Edit Property Set** dialog box, click **Options > Property Sets > Edit**.

The dialog boxes contain the following items:

Name	Type the name of the property set you want to create. Property set names are not case sensitive and they can be any length and consist of any combination of alphanumeric characters and underscore characters.
List of property sets	Displays existing property sets. This list changes, depending on whether you are adding new property sets or editing them.
Class	Type the name of the property set's class. The name of the class is not validated, so be sure to type carefully. Invalid names are ignored at runtime.
Property	Type the property of the class you have just identified.
Add	Click to add the class/property pair to the list.
Edit	Click after selecting a class/property set you want to edit.
Remove	Click after selecting a class/property set you want to delete.

Combine Property Sets Dialog Box

This functionality is supported only if you are using the Classic Agent.

Use the **Combine Property Sets** dialog box to combine existing property sets.

To open the dialog box, click **Options > Property Sets > Combine**. You must select an existing property set before **Combine** is enabled.

The **Combine Property Sets** dialog box includes the following items:

Name	Specify a name for the property set you are creating.
Property sets to combine	From this list, select at least two property sets that you want to combine into a new property set.



Note: If any of the existing property sets are modified, the combined property set will be modified as well.

DOM Extensions Dialog Box

Use to set object and table recognition options for the DOM extension.

Click **Options > Extensions**, enable a primary extension in the **Primary Extension** column, and then click **Extension** in the **Options** area.



Note: The primary extension must be enabled before you can click **Extension**.

The information that you enter on this dialog box is saved in the `domex.ini` file in your `<SilkTest installation directory>\Extend` directory. To avoid any confusion, we recommend that you do not have the `domex.ini` file open while you are working with the **DOM Extensions** dialog box.

There are several DOM extension options that are not available through the **DOM Extensions** dialog box.

Show the following components area

Borderless Table	Check the Table check box and set a value for borderless table recognition. .76 is the threshold where Silk Test Classic starts to recognize more objects within tables, such as images, hidden text, check boxes, textfields, and buttons.
Meta	If checked, the DOM extension records Meta objects. Default is checked. Uncheck this check box if you do not want the DOM extension to record Meta objects.
Hidden	If checked, the DOM extension records Hidden objects. Default is checked. Uncheck this check box if you do not want the DOM extension to record Hidden objects.
XML	If checked, the DOM extension records XML objects. Default is unchecked. Check this check box if you want the DOM extension to record XML objects.
Invisible	If checked, the DOM extension records invisible objects. Default is checked. Uncheck this check box if you do not want the DOM extension to record invisible objects. If your browser-based application consists of pages that contain many invisible objects that you do not need to test, then you can improve performance by un-checking this check box in order to ignore all invisible objects.
Form	If checked, the DOM extension records Form objects. Default is unchecked. Check this check box if you want the DOM extension to record Form objects.
Text	If checked, the DOM extension records Text objects. Default is checked. Uncheck this check box if you do not want the DOM extension to record Text objects. If you are testing a transaction type page with a lot of text, consider un-checking the Text check box. This prevents Silk Test Classic from recording many text objects, which helps your declarations to be clean. If, on the other hand, you're looking for formatting and styles of text objects, you'll want to select this option.
BodyText	If checked, the DOM extension records BodyText objects, which is text that is not contained within an HTML tag. (In previous releases, body text appeared as HtmlText.) Default is unchecked. We suggest keeping this option unchecked for improved performance, particularly when recording window declarations on large pages. Should you need to record body text, check this check box. Silk Test Classic sometimes treats body text differently from HTML text that is in an HTML tag. For example, <code>GetChildren()</code> will not return body text. Also, Record Window Declarations will not highlight the window declaration of a body text object when the cursor is moved over it, and other recorders such as Record Window Identifiers will not detect it. <code>IsVisible()</code> returns <code>FALSE</code> for body text.
List Item	If checked, the DOM extension shows the text contained within <code>HtmlList</code> controls in your browser. Default is checked. If mouse events are associated with your list items, check this

check box so Silk Test Classic can interact with the list items. This setting is global. However, if you want to set this option for only certain points in your script, use `BrowserPage.SetUserOption()` as described in `SetUserOption()`.

Object recognition options area

Use Browser nearest text	Determines how the DOM extension finds the closest static text for <code>HtmlTable</code> , <code>HtmlLink-text</code> , <code>HtmlColumn</code> , <code>HtmlLink-image</code> , <code>HtmlImage</code> , <code>HtmlHeading</code> , <code>HtmlText</code> , <code>HtmlRadioList</code> , and <code>HtmlPushbutton</code> . Check this check box if you want Silk Test Classic to use the DOM extension to find the closest static text for the objects listed above. This does not apply to invisible objects such as XML, Meta, and Hidden; those objects do not rely on any text on a page and so it would be meaningless to try to associate them with any objects. Uncheck this check box if you want to use the Agent to determine closest static text for the objects listed above.
Use virtual column	<p>Affects how the DOM extension records asymmetric tables. These are tables that use either column span or row span attributes, or tables whose rows don't have the same number of columns. An example of an asymmetric table is a typical calendar page that has the month of January written across the top row and the seven days of the week in seven columns across the 2nd row.</p> <p>We highly recommend that you check this box if you are working with tables that have asymmetrical rows.</p> <p>Check this check box if you want to create virtual columns for any row in a table. In the example above, it causes the top row to contain one real column for "January", followed by 6 virtual columns which are blank and align with Mon, Tues, etc. These virtual columns appear where there are none in order to complete the table and they are named <code>virtual1</code>, <code>virtual2</code>, and so on. These virtual columns cause the table to be symmetrical.</p> <p>Uncheck this check box to avoid creating virtual columns. This causes Silk Test Classic to record the top row as the name for first column. This occurs because there is no 2nd column in the top row; Mon is promoted to the name of the second column, and so on.</p>
Search whole DOM tree	Determines how windows declarations are found. To search the entire DOM tree when you record windows declarations, check the Search whole DOM tree check box.

Extension Application Dialog Box

This functionality is supported only if you are using the Classic Agent.

Use the **Extension Application** dialog box to add one or more runtime environments to the **Extensions** dialog box for testing applications.

Click **Options > Extensions** and then click **New**.

Enter name(s) of executable or DLL file	Type the name of the executable or DLL that you want to add, separated by commas. If the executable name contains spaces, enclose the name in quotation marks. Note that after enabling a new application, you may need to run the Extension Enabler on the target machine and restart any applications affected by these changes.
Browse button	Click to browse to the executable you want to add.

Extension Options (ActiveX) Dialog Box

This functionality is supported only if you are using the Classic Agent.

Use to specify options for the currently selected extension.

Click **Options** > **Extensions**, select the appropriate extension, enable ActiveX in the **Other Extensions** column, and then click **ActiveX**.

Ignore VB and SS Frames

Check this check box to allow scripts created prior to Silk Test Classic 5.0 to run properly. When this option is selected, the frame is essentially ignored and objects in VB and SS (Sheridan OCX) frames are treated as children of the main window rather than child objects of the containing frame. If you do not select this option for scripts created prior to Silk Test Classic 5.0, you will need to re-record window declarations in the current release and edit your scripts.



Note: If the controls inside the VB and SS frames are windowless, then this option will not work properly. In this case, you can use this option successfully only if you change the controls so that they have windows.

Do not select this option for window declarations recorded in Silk Test Classic 5.0 or later, which treats objects in these frames as children of the frame and grandchildren of the main window. The window declarations file reflects this deeper nesting.



Note: This change brings Silk Test Classic into conformance with current MS Windows architecture.

Extension Options Dialog Box (Java)

This functionality is supported only if you are using the Classic Agent.

Use the **Extension Options** dialog box for Java to specify options for the currently selected extension.

Click **Options** > **Extensions**, select the appropriate extension, and then click **Java**.

Java console filename

Type the path to the file to which you want to redirect Java console output.

Redirect Java console output

Check this check box if you want to redirect output from the Java console to a local file where you can easily scroll and copy the text. When testing with the SilkBean, you cannot redirect console output using this option.

TrueLog Options - Classic Agent Dialog Box

Use the **TrueLog Options - Classic Agent** dialog box to enable TrueLog for the Classic Agent and to customize the information that the TrueLog collects for Silk Test Classic.

Click **Options** > **TrueLog**.

Logging bitmaps and controls in TrueLog may adversely affect the performance of Silk Test Classic. Because capturing bitmaps and logging information can result in large TrueLog files, you may want to log test cases with errors only and then adjust the TrueLog options for test cases where more information is needed.



Note: By default, TrueLog Explorer is enabled when you are using the Open Agent, and disabled when you are using the Classic Agent. When TrueLog Explorer is enabled, the default setting is that

screenshots are only created when an error occurs in the script and only test cases with errors are logged.

For additional information about TrueLog Explorer, refer to the *Silk TrueLog Explorer User Guide*, located in (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Documentation** or (in Microsoft Windows 10) **Start > Silk**.

Logging Settings tab

Enables the TrueLog capture and configures the TrueLog.

Enable TrueLog

Captures TrueLog data and activates logging settings. You can also enable or disable TrueLog each time you run a test case by checking this check box in the **Run Testcase** dialog box. When you enable or disable TrueLog in the **Run Testcase** dialog box, Silk Test Classic makes the same change in the **TrueLog Options** dialog box. Likewise, when you enable or disable TrueLog in the **TrueLog Options** dialog box, Silk Test Classic makes the same change in the **Run Testcase** dialog box.

All testcases Logs activity for all test cases, both successful and failed.

Testcases with errors Logs activity for only those test cases with errors.

TrueLog file

Designates the location and name of the TrueLog file. This path is relative to the machine on which the agent is running. The name defaults to the name used for the results file, with an `.xlg` extension. The location defaults to the same folder as the test case `.res` file.



Note: If you provide a local or remote path in this field, the path cannot be validated until script execution time.

Log the following for controls

Logs certain types of information about the controls on the active window or page.

Control information

Logs the GUI control's hierarchy, name, type, and other attributes for the active window or page. This information appears on the **Controls** tab of the **Information** window in TrueLog Explorer. This is selected by default when you enable TrueLog. If you choose to log control information, you may optionally decide to log:

Control creation/deletion Tracks the creation and/or deletion of controls on the active window or page. TrueLog updates the control hierarchy after each action. Choosing this option may adversely affect performance.

Include static text controls Includes the static text controls in the logged hierarchy. To keep TrueLogs small, leave this option turned off for browser testing.

Track low level events Logs keyboard and mouse events. For each mouse click and key press a new **Action** node is created on the Tree List in the TrueLog. Use caution when selecting this option for browser applications, as you can significantly impact performance.

Log the following for browsers

Identifies the browser events you want to capture.

Download events Captures all events that trigger page downloads. Selected by default.

Navigate events Captures all events that cause new pages to appear. Selected by default.

Terminate events Captures all events that cause a browser to close. Selected by default.

New window events	Captures all events that cause a new browser window to appear. Selected by default.
MouseMove calls	Logs all scripted <code>MouseMove</code> calls. This is useful for tracking JavaScript <code>MouseOver</code> events. By including this information in TrueLogs, you may significantly affect performance. This information appears as an Action node in the Tree List.

TrueLog delay Allows Windows time to draw the application window before a bitmap is taken. The delay can be used for browser testing. You can insert a `Browser.WaitForReady` call in your script to ensure that the `DocumentComplete` events are seen and processed. If **WindowActive** nodes are missing from the **TrueLog**, you need to add a `Browser.WaitForReady` call. You can also use `TrueLog Delay` to optimize script performance. Set the delay as small as possible to get the correct behavior and have the smallest impact on script execution time. The default setting is 0.

Enable bitmap capture Controls when TrueLog captures screenshots of the application under test. Bitmap files are included in TrueLog (.xlg) files. Selected by default.

As specified on action settings tab	Enables the capture of bitmaps for each action type you select on the Action Settings tab. This setting also causes a bitmap capture each time a window becomes active in your application.
On error	Captures a bitmap when an error occurs in your test case.
Window only	Saves the bitmaps for the active window.
Desktop	Saves a bitmap of the entire desktop.
Before error bitmap	Captures the bitmap before the error occurred.

TrueLog Presets Sets pre-determined logging levels.

Minimal	Logs test cases with errors; enables bitmap capture of desktop on error; does not log any actions.
Default	Logs test cases with errors; enables bitmap capture of window on error; logs data for <code>Select</code> and <code>SetText</code> actions; enables bitmap capture for <code>Select</code> and <code>SetText</code> actions.
Full	Logs all test cases; logs all control information; logs all events for browsers except for <code>MouseMove</code> events; enables bitmap capture of the window on error; captures bitmaps for all actions.

If you enable Full logs and encounter a `Window Not Found` error, you may need to manually edit your script.

Action Settings tab

The **Action Settings** tab on the **TrueLog Options - Classic Agent** dialog box selects the scripted actions you want to include in the TrueLog. When enabled, these actions appear as nodes in the Tree List view of the TrueLog.

Select actions to log	Enable	Selects the action to log. Each action corresponds to a 4Test method, except for <code>Click</code> and <code>Select</code> .
------------------------------	---------------	---

- Click** Records mouse clicks on many controls, such as `PushButton`, `ScrollBar`, `TextField`, and `HtmlLink`. To record `Click` methods on a `CheckBox`, choose `Select`, not `Click`.
- Select** Records actions on multiple methods of multiple types of controls, including `ListBox`, `TreeView`, `ComboBox`, `RadioButton`, and `CheckBox`. Here is a partial list of what `Select` records:
- **Select (ListBox, TreeView, ComboBox, RadioButton)**
 - **DoubleSelect, SelectList, SelectRange (ListBox, TreeView)**
 - **Click (ListBox, TreeView, ComboBox, RadioButton, CheckBox)**
 - **Check, Uncheck, Toggle, SetState (CheckBox)**

Bitmap Selects the point in time you want bitmaps to be captured:

- None** Never captures bitmaps.
- Before** Captures bitmaps before errors occur.
- After** Captures bitmaps after errors occur.
- Both** Capture bitmaps both before and after errors occur.

TrueLog Options - Open Agent Dialog Box

Use the **TrueLog Options - Open Agent** dialog box to enable TrueLog for the Open Agent and to customize the information that the TrueLog collects for Silk Test Classic.

Click **Options > TrueLog**.

You can additionally use the **Difference Viewer** to analyze results for test cases that use the Open Agent.

Logging bitmaps and controls in a TrueLog may adversely affect the performance of Silk Test Classic. Because capturing bitmaps and logging information can result in large TrueLog files, you may want to log test cases with errors only and then adjust the TrueLog options for test cases where more information is needed.



Note: By default, TrueLog Explorer is enabled when you are using the Open Agent, and disabled when you are using the Classic Agent. When TrueLog Explorer is enabled, the default setting is that screenshots are only created when an error occurs in the script and only test cases with errors are logged.

For additional information about TrueLog Explorer, refer to the *Silk TrueLog Explorer User Guide*, located in (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Documentation** or (in Microsoft Windows 10) **Start > Silk**.

Enable TrueLog Captures TrueLog data and activates logging settings. You can also enable or disable TrueLog each time you run a test case by checking this check box in the **Run Testcase** dialog box. When you enable or disable TrueLog in the **Run Testcase** dialog box, Silk Test Classic makes the same change in the **TrueLog Options** dialog box. Likewise, when you enable or disable TrueLog in the **TrueLog Options** dialog box, Silk Test Classic makes the same change in the **Run Testcase** dialog box.

All testcases Logs activity for all test cases, both successful and failed.

Testcases with errors Logs activity for only those test cases with errors.

TrueLog Location

The path is relative to the machine on which the agent is running. The default path is the path of the Silk Test Classic project folder, and the default name is the name of the suite class, with a `.tlz` suffix. To ensure that TrueLog files are not overwritten, for example when you perform parallel testing, you can add placeholders to the TrueLog file name. These placeholders are replaced with the appropriate data at execution time.



Note: The path is validated at execution time. Tests that are executed by Silk Central set this value to the Silk Central results directory to enable the screenshots to be shown in the result view.

Screenshot mode

Controls when TrueLog Explorer captures test application screens during playback of the application under test. Bitmap files are included in TrueLog (`.xlg`) files.



Note: For test cases with errors, TrueLog Explorer always captures a screen, independent of the screenshot mode you have selected.

- None** Captures no screens during playback.
- Active Window** Captures only the active window of the test application during playback.
- Active Application** Captures the test application and any windows within the test application.
- Desktop** Captures the entire desktop area.

Delay

Allows Windows time to draw the application window before a bitmap is taken. You can also use `Delay` to optimize script performance. Set the delay as small as possible to get the correct behavior and have the smallest impact on script execution time. The default setting is 0.

Troubleshooting the Open Agent

This section provides information and workarounds for working with the Open Agent.

Troubleshooting Apache Flex Applications

This functionality is supported only if you are using the Open Agent.

This section provides help and troubleshooting information for working with Apache Flex applications.

Why Cannot Silk Test Classic Recognize Apache Flex Controls?

This functionality is supported only if you are using the Open Agent.

If Silk Test Classic cannot recognize the controls of an Apache Flex application, which you are accessing through a Web server, you can try the following things:

- Compile your Apache Flex application with the Adobe automation libraries and the appropriate `FlexTechDomain.swc` for the Apache Flex version.
- Use runtime loading.
- Apache Flex controls are not recognized when embedding an Apache Flex application with an empty `id` attribute.

Troubleshooting Basic Workflow Issues with the Open Agent

The following troubleshooting tips might help you with the basic workflow:

Why is my application not displayed in the Select Application dialog box?

If the application that you want to test is not displayed in the **Select Application** dialog box, try the following:

- Uncheck the **Hide processes without caption** check box. This check box is checked by default and prevents applications without a caption from being displayed in the dialog box.
- Run Silk Test Classic with elevated privileges.
 1. Close Silk Test Classic.
 2. Stop the Open Agent.
 3. Run Silk Test Classic as an administrator.
- Use the **Task Manager** to check if the application is running under a different user account.
- Ensure that the application is not started with the `runas` command or a similar command.

Error Messages

This section provides help and troubleshooting information for error messages.

Agent not responding

Problem

You get the following error message:

```
Error: Agent not responding
```

This error can occur for a number of reasons.

Solution

Try any or all of the following:

- Restart the application that you are testing.
- Restart Silk Test Classic.
- Restart the Host machine.

If you are recording declarations on a very large page and get this error, consider increasing the `AgentTimeout`.

Control is not responding

Problem

You run a script and get the following error: `Error: Control is not responding`

This is a catch-all error message. It usually occurs in a `Select()` statement when Silk Test Classic is trying to select an item from a `ListBox`, `TreeView`, `ListView`, or similar control.

The error can occur after the actual selection has occurred, or it can occur without the selection being completed. In general the error means that the object is not responding to the messages Silk Test Classic is sending in the manner in which it expects.

Solution

Try these things to eliminate the error message:

- If the line of code is inside a `Recording` block, remove the `Recording` keyword.
- Set the following option just before the line causing the error:
`Agent.SetOption(OPT_VERIFY_RESPONDING, FALSE).`
- If the selection is successful, but you still get the error, try using the `Do . . . except` feature.

Functionality Not Supported on the Open Agent

If you use Classic Agent functionality in an Open Agent script, an error message displays, stating that the functionality is not supported on the Open Agent.

Example

For example, if you try to call the `ClearTrap` function of the Classic Agent on a `MainWin` object in an Open Agent script, the following error message displays:

```
The Open Agent does not support the function  
'MainWin::ClearTrap' #
```


Unable to Connect to Agent

Problem

You get the following error message: `Error: Unable to connect to agent`

This error can occur for a number of reasons.

Solution

Connect to the default agent Click **Tools > Connect to Default Agent**.

The command starts the Classic Agent or the Open Agent on the local machine depending on which agent is specified as the default in the **Runtime Options** dialog. If the agent does not start within 30 seconds, a message is displayed. If the default agent is configured to run on a remote machine, you must connect to it manually.

Restart the agent that you require for testing Click (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Tools > Silk Test Open Agent** or (in Microsoft Windows 10) **Start > Silk > Silk Test Open Agent** or (in Microsoft Windows 7) **Start > Programs > Silk > Silk Test > Tools > Silk Test Classic Agent** or (in Microsoft Windows 10) **Start > Silk > Silk Test Classic Agent**.

Window is not active

Problem

You run a script and get the following error: `Error: Window 'name' is not active.`

This error means that the object Silk Test Classic is trying to act on is not active. This message applies to top-level windows (`MainWin`, `DialogBox`, `ChildWin`).

Solution

You can correct the error by doing one of the following:

1. Edit the script and add an explicit `SetActive()` statement to the window you are trying to act on just above the line where the error is occurring. An easy way to do this is to double-click the error in the results file. You will be brought to the line in the script. Insert a new line above it and add a line ending with the `SetActive()` method.

2. Tell Silk Test Classic not to verify that windows are active. There are two ways to do this:

To turn off the verification globally, uncheck the **Verify that windows are active** option on the **Verification** tab in the **Agent Options** dialog (**Options > Agent**).

To turn off the option in your script on a case by case basis, add the following statement to the script, just before the line causing the error: `Agent.SetOption(OPT_VERIFY_EXPOSED, FALSE)`.

3. Then add the following line just after the line causing the error:

```
Agent.SetOption(OPT_VERIFY_EXPOSED, TRUE).
```

This means Silk Test Classic will execute the action regardless of whether the window is active.

4. Extend the window time out to be greater than 10 by inserting the **Agent - Window Timeout** to `>= 10` into your `partner.ini`.

Window is not enabled

Problem

You run a script and get the following error: `Error: Window 'name' is not enabled.`

This error means that the object that Silk Test Classic is trying to act on is not enabled. This message applies to controls inside top-level windows (such as `PushButton` and `CheckBox`).

Solution

You can correct this problem in one of two ways.

- If the object is indeed disabled, edit the script and add the actions that will enable the object.
- If the object is in fact enabled and you want the script to perform the action, tell Silk Test Classic not to verify that a window is enabled:

To turn off the verification globally, uncheck the **Verify that windows are enabled** option on the **Verification** tab in the **Agent Options** dialog box (**Options > Agent**).

To turn off the option in your script on a case-by-case basis, add the following statement to the script, just before the line causing the error: `Agent.SetOption(OPT_VERIFY_ENABLED, FALSE)`

Then add the following line just after the line causing the error:

```
Agent.SetOption(OPT_VERIFY_ENABLED, TRUE).
```

This means Silk Test Classic will execute the action regardless of whether the window is enabled.

Window is not exposed

Problem

You run a script and get the following error: `Error: Window 'name' is not exposed.`

Sometimes, applications are written such that windows are hidden to the operating system, even though they are fully exposed to the user. A running script might generate an error such as `Window not exposed`, even though you can see the window as the script runs.

Solution

While it might be tempting to simply turn off the checks for these verifications from the **Agent Options > Verification** dialog box, the best course of action is to take such errors on a case by case basis, and only turn off the verification in cases where the window is genuinely viewable, but Silk Test Classic is getting information from the operating system saying the object is not visible.

1. Add the following statement to the script, just before the line causing the error:
`Agent.SetOption(OPT_VERIFY_EXPOSED, FALSE).`
2. Then add the following line just after the line causing the error:
`Agent.SetOption(OPT_VERIFY_EXPOSED, TRUE).`

This means Silk Test Classic will execute the action regardless of whether it thinks the window is exposed.

Window not found

Problem

You run a script and get the following error: `Error: Window 'name' was not found.`

Resolution

This error occurs in the following situations:

When the window that Silk Test Classic is trying to perform the action on is not on the desktop.

If you are watching the script run, and at the time the error occurs you can see the window on the screen, it usually means the tag that was generated is not a correct tag. This could happen if the application changed from the time the script or include file was originally created.

To resolve this issue, enable view trace listing in your script.

The window is taking more than the number of seconds specified for the window timeout to open.

To resolve this issue, set the **Window Timeout** value to prevent `Window Not Found` exceptions

Only if you are using the Classic Agent, in the TrueLog Options - Classic Agent dialog box, if all of the following options are set

- The action `PressKeys` is enabled.
- Bitmaps are captured after or before and after the `PressKeys` action.
- `PressKeys` actions are logged.

The preceding settings are set by default if you select `Full` as the **TrueLog** preset.

To resolve this issue, modify your test case.

Handling Exceptions

This section provides help and troubleshooting information for handling exceptions.

Default Error Handling

If a test case fails, for example if the expected value doesn't match the actual value in a verification statement, by default Silk Test Classic calls its built-in recovery system, which:

- Terminates the test case.
- Logs the error in the results file.
- Restores your application to its default base state in preparation for the next test case.

These runtime errors are called exceptions. They indicate that something did not go as expected in a script. They can be generated automatically by Silk Test Classic, such as when a verification fails, when there is a division by zero in a script, or when an invalid function is called.

You can also generate exceptions explicitly in a script.

However, if you do not want Silk Test Classic to transfer control to the recovery system when an exception is generated, but instead want to trap the exception and handle it yourself, use the `4Test do . . . except` statement.

Custom Error Handling

You can also use `do . . . except` to perform some custom error handling, then use the re-raise statement to pass control to the recovery system as usual.

Example: do ... except

The Text Editor application displays a message box if a user searches for text that does not exist in the document. You can create a data-driven test case that verifies that the message box appears and that it displays the correct message. Suppose you want to

determine if the Text Editor application is finding false matches, that is, if it is selecting text in the document before displaying the message box. That means that you want to do some testing after the exception is raised, instead of immediately passing control to the recovery system. The following code sample shows how you can use `do ... except` to keep the control inside the test case:

```
testcase Negative (SEARCHINFO Data)
  STRING sMatch
  TextEditor.File.New.Pick ()
  DocumentWindow.Document.TypeKeys (Data.sText + Data.sPos)
  TextEditor.Search.Find.Pick ()
  Find.FindWhat.SetText (Data.sPattern)
  Find.CaseSensitive.SetState (Data.bCase)
  Find.Direction.Select (Data.sDirection)
  Find.FindNext.Click ()

  do
    MessageBox.Message.VerifyValue (Data.sMessage)
  except
    sMatch = DocumentWindow.Document.GetSelText ()

    if (sMatch != "")
      Print ("Found " + sMatch + " not " + Data.sPattern)
      reraise
    MessageBox.OK.Click ()

  Find.Cancel.Click ()
  TextEditor.File.Close.Pick ()
  MessageBox.No.Click ()
```

This following tasks are performed in the example:

- A test is performed after an exception is raised.
- A statement is printed to the results file if text was selected.
- The recovery system is called.
- The recovery system terminates the test case, logs the error, and restores the test application to its default base state.

As the example shows, following the `do` keyword is the verification statement, and following the `except` keyword are the 4Test statements that handle the exception. The exception-handling statements in this example perform the following tasks:

- Call the `GetSelText` method to determine what text, if any, is currently selected in the document.
- If the return value from the `GetSelText` method is not an empty string, it means that the application found a false match.
- If the application found a false match, print the false match and the search string to the results file.
- Re-raise the exception to transfer control to the recovery system.
- Terminate the test case.

The `reraise` statement raises the most recent exception again and passes control to the next exception handler. In the preceding example, the `reraise` statement passes control to the built-in recovery system. The `reraise` statement is used in the example because if the exception-handling code does not explicitly re-raise the exception, the flow of control passes to the next statement in the test case.

Trapping the exception number

Each built-in exception has a name and a number (they are defined as an enumerated data type, EXCEPTION). For example, the exception generated when a verify fails is `E_VERIFY (13700)`, and the exception generated when there is a division by zero is `E_DIVIDE_BY_ZERO (11500)`.

All exceptions are defined in `4test.inc`, in the directory where you installed Silk Test Classic.

You can use the `ExceptNum` function to test for which exception has been generated and, perhaps, take different actions based on the exception. You would capture the exception in a `do...except` statement then check for the exception using `ExceptNum`.

For example, if you want to ignore the exception `E_WINDOW_SIZE_INVALID`, which is generated when a window is too big for the screen, you could do something like this:

```
do
Open.Invoke ()
except
if (ExceptNum () != E_WINDOW_SIZE_INVALID)
    reraise
```

If the exception is not `E_WINDOW_SIZE_INVALID`, the exception is reraised (and passed to the recovery system for processing). If the exception is `E_WINDOW_SIZE_INVALID`, it is ignored.

Defining Your Own Exceptions

In addition to using built-in exceptions, you can define your own exceptions and generate them using the `raise` statement.

Consider the following test case:

```
testcase raiseExample ()
    STRING sTestValue = "xxx"
    STRING sExpected = "yyy"
    TestVerification (sExpected, sTestValue)

TestVerification (STRING sExpected, STRING sTestValue)
    if (sExpected == sTestValue)
        Print ("Success!")
    else
        do
            raise 1, "{sExpected} is different than {sTestValue}"
        except
print ("Exception number is {ExceptNum()}")
    reraise
```

The `TestVerification` function tests two strings. If they are not the same, they raise a user-defined exception using the `raise` statement.

Raise Statement

The `raise` statement takes one required argument, which is the exception number. All built-in exceptions have negative numbers, so you should use positive numbers for your user-defined exceptions. `raise` can also take an optional second argument, which provides information about the exception; that information is logged in the results file by the built-in recovery system or if you call `ExceptLog`.

In the preceding test case, `raise` is in a `do...except` statement, so control passes to the `except` clause, where the exception number is printed, then the exception is reraised and passed to the recovery system, which handles it the same way it handles built-in exceptions.

Here is the result of the test case:

```
Testcase raiseExample - 1 error
Exception number is 1
yyy is different than xxx
Occurred in TestVerification at except.t(31)
Called from raiseExample at except.t(25)
```

Note that since the error was re-raised, the test case failed.

Using do...except Statements to Trap and Handle Exceptions

Using `do...except` you can handle exceptions locally, instead of passing control to the built-in error handler, which is part of the recovery system of Silk Test Classic. The `do...except` statement has the following syntax:

```
do
<statements>
except
<statements>
```

If an exception is raised in the `do` clause of the statement, control is immediately passed to the `except` clause, instead of to the recovery system.

If no exception is raised in the `do` clause of the statement, control is passed to the line after the `except` clause. The statements in the `except` clause are not executed.

Consider this simple test case:

```
testcase except1 (STRING sExpectedVal, STRING sActualVal)
do
  Verify (sExpectedVal, sActualVal)
  Print ("Verification succeeded")
except
  Print ("Verification failed")
```

This test case uses the built-in function `Verify`, which generates an exception if its two arguments are not equivalent. In this test case, if `sExpectedVal` equals `sActualVal`, no exception is raised, `Verification succeeded` is printed, and the test case terminates. If the two values are not equal, `Verify` raises an exception, control immediately passes to the `except` clause, the first `Print` statement is not executed, and `Verification failed` is printed.

Here is the result if the two values "one" and "two" are passed to the test case:

```
Testcase except1 ("one", "two") - Passed
Verification failed
```

The test case passes and the recovery system is not called because you handled the error yourself.

You handle the error in the `except` clause. You can include any 4Test statements, so you could, for example, choose to ignore the error, write information to a separate log file, and log the error in the results file.

Programmatically Logging an Error

Test cases can pass, even though an error has occurred, because they used their own error handler and did not specify to log the error. If you want to handle errors locally and generate an error (that is, log an error in the results file), you can do any of the following:

- After you have handled the error, re-raise it using the `reraise` statement and let the default recovery system handle it.

- Call any of the following functions in your script:

LogError (string, [cmd-line])	Writes string to the results file as an error (displays in red or italics, depending on platform) and increments the error counter. This function is called automatically if you don't handle the error yourself. cmd-line is an optional string expression that contains a command line.
LogWarning (string)	Same as LogError, except it logs a warning, not an error.
ExceptLog ()	Calls LogError with the data from the most recent exception.

Performing More than One Verification in a Test Case

If the verification fails in a test case with only one verification statement, usually an exception is raised and the test case is terminated. However, if you want to perform more than one verification in a test case, before the test case terminates, this approach would not work.

Classic Agent Example

For example, see the following sample test case:

```
testcase MultiVerify ()
  TextEditor.Search.Find.Pick ()
  Find.VerifyCaption ("Find")
  Find.VerifyFocus (Find.FindWhat)
  Find.VerifyEnabled (TRUE)
  Find.Cancel.Click ()
```

The test case contains three verification statements. However, if the first verification, `VerifyCaption`, fails, an exception is raised and the test case terminates. The second and the third verification are not executed.

To perform more than one verification in a test case, you can trap all verifications except the last one in a `do . . . except` statement, like the following sample for the Classic Agent shows:

```
testcase MultiVerify2 ()
  TextEditor.Search.Find.Pick ()
  do
    Find.VerifyCaption ("Find")
  except
    ExceptLog ()
  do
    Find.VerifyFocus (Find.FindWhat)
  except
    ExceptLog ()
  Find.VerifyEnabled (TRUE)
  Find.Cancel.Click ()
```

All the verifications in this example are executed each time that the test case is run. If one of the first two verifications fails, the 4Test function `ExceptLog` is called. The `ExceptLog` function logs the error information in the results file, then continues the execution of the script.

Open Agent Example

For example, you might want to print the text associated with the exception as well as the function calls that generated the exception. The following test case illustrates this:

```
testcase VerifyTest ()
  STRING sTestValue = "xxx"
  STRING sExpectedValue = "yyy"
  CompValues (sExpectedValue, sTestValue)

CompValues (STRING sExpectedValue, STRING sTestValue)
do
  Verify (sExpectedValue, sTestValue)
except
  ErrorHandler ()

ErrorHandler ()
  CALL Call
  LIST OF CALL lCall
  lCall = ExceptCalls ()
  Print (ExceptData ())
  for each Call in lCall
    Print("Module: {Call.sModule}",
          "Function: {Call.sFunction}",
          "Line: {Call.iLine}")
```

- The test case calls the user-defined function `CompValues`, passing two arguments.
- `CompValues` uses `Verify` to compare its arguments. If they are not equal, an exception is automatically raised.
- If an exception is raised, `CompValues` calls a user-defined function, `ErrorHandler`, which handles the error. This is a general function that can be used throughout your scripts to process errors the way you want.
- `ErrorHandler` uses two built-in exception functions, `ExceptData` and `ExceptCalls`.

Except Data All built-in exceptions have message text associated with them. `ExceptData` returns that text.

ExceptCalls Returns a list of the function calls that generated the exception. You can see from `ErrorHandler` above, that `ExceptCalls` returns a `LIST OF CALL`. `CALL` is a built-in data type that is a record with three elements:

- `sFunction`
- `sModule`
- `iLine`

`ErrorHandler` processes each of the calls and prints them in the results file.

- Silk Test Classic also provides the function `ExceptPrint`, which combines the features of `ExceptCalls`, `ExceptData`, and `ExceptNum`.

```
Testcase VerifyTest - Passed
*** Error: Verify value failed - got "yyy", expected "xxx"
Module: Function: Verify Line: 0
Module: except.t Function: CompValues Line: 121
Module: except.t Function: VerifyTest Line: 112
```

The second line is the result of printing the information from `ExceptData`. The rest of the lines show the processing of the information from `ExceptCalls`.

This test case passes because the error was handled locally and not re-raised.

Writing an Error-Handling Function

If you want to customize your error processing, you will probably want to write your own error-handling function, which you can reuse in many scripts.

Open Agent Example

For example, you might want to print the text associated with the exception as well as the function calls that generated the exception. The following test case illustrates this:

```
testcase VerifyTest ()
  STRING sTestValue = "xxx"
  STRING sExpectedValue = "yyy"
  CompValues (sExpectedValue, sTestValue)

CompValues (STRING sExpectedValue, STRING sTestValue)
do
  Verify (sExpectedValue, sTestValue)
except
  ErrorHandler ()

ErrorHandler ()
  CALL Call
  LIST OF CALL lCall
  lCall = ExceptCalls ()
  Print (ExceptData ())
  for each Call in lCall
    Print("Module: {Call.sModule}",
          "Function: {Call.sFunction}",
          "Line: {Call.iLine}")
```

- The test case calls the user-defined function `CompValues`, passing two arguments.
- `CompValues` uses `Verify` to compare its arguments. If they are not equal, an exception is automatically raised.
- If an exception is raised, `CompValues` calls a user-defined function, `ErrorHandler`, which handles the error. This is a general function that can be used throughout your scripts to process errors the way you want.
- `ErrorHandler` uses two built-in exception functions, `ExceptData` and `ExceptCalls`.

Except Data All built-in exceptions have message text associated with them. `ExceptData` returns that text.

ExceptCalls Returns a list of the function calls that generated the exception. You can see from `ErrorHandler` above, that `ExceptCalls` returns a `LIST OF CALL`. `CALL` is a built-in data type that is a record with three elements:

- `sFunction`
- `sModule`
- `iLine`

`ErrorHandler` processes each of the calls and prints them in the results file.

- Silk Test Classic also provides the function `ExceptPrint`, which combines the features of `ExceptCalls`, `ExceptData`, and `ExceptNum`.

```
Testcase VerifyTest - Passed
*** Error: Verify value failed - got "yyy", expected "xxx"
Module: Function: Verify Line: 0
Module: except.t Function: CompValues Line: 121
Module: except.t Function: VerifyTest Line: 112
```

The second line is the result of printing the information from `ExceptData`. The rest of the lines show the processing of the information from `ExceptCalls`.

This test case passes because the error was handled locally and not re-raised.

Exception Values

This section describes the exceptions that are generated by Silk Test Classic under specific error conditions.

Exception value	Description
E_ABORT	Script aborted by user.
E_APP_NOT_READY	The application is not ready.
E_APP_NOT_RESPONDING	The application is not responding to input.
E_APPID_INVALID	The specified application ID is not a valid application.
E_BITMAP_NOT_STABLE	The bitmap timeout period set with <code>OPT_BITMAP_MATCH_TIMEOUT</code> was reached before the image stabilized.
E_BITMAP_REGION_INVALID	The specified region was off the screen.
E_BITMAPS_DIFFERENT	The comparison failed when comparing two bitmaps.
E_CANT_CLEAR_SELECTION	The selection cannot be cleared.
E_CANT_CLOSE_WINDOW	The window cannot be closed (often resulting when a confirmation dialog box pops up).
E_CANT_COMPARE_BITMAP	Silk Test Classic ran out of a system resource (such as memory) needed to compare the bitmaps.
E_CANT_CONVERT_RESOURCE	The specified resource cannot be handled by <code>GetResource</code> , although it is a valid resource for the widget.
E_CANT_EXIT_APP	Silk Test Classic was unable to close the application.
E_CANT_EXTEND_SELECTION	The list box selection can not be extended because nothing is selected.
E_CANT_MAXIMIZE_WINDOW	The window can not be maximized.
E_CANT_MINIMIZE_WINDOW	The window can not be minimized.
E_CANT_MOVE_WINDOW	The window can not be moved.
E_CANT_RESTORE_WINDOW	The window size can not be restored.
E_CANT_SET_ACTIVE	The window can not be set active.
E_CANT_SET_FOCUS	The window can not be given the input focus.
E_CANT_SIZE_WINDOW	The window can not be resized.

Exception value	Description
E_CANT_START_APP	The application cannot be started.
E_COL_COUNT_INVALID	The specified value is not a valid character count.
E_COL_NUM_INVALID	The specified value is not a valid character position.
E_COL_START_EXCEEDS_END	The starting character exceeds the end character position.
E_COLUMN_INDEX_INVALID	The specified index is not a valid column index. All <i>DataGrid</i> methods that use <i>DataGridCell</i> , <i>DataGridRow</i> , or <i>DataGridColumn</i> as a parameter may see this exception raised.
E_COLUMN_NAME_INVALID	The specified index is not a valid column index. All <i>DataGrid</i> methods that use <i>DataGridCell</i> , <i>DataGridRow</i> , or <i>DataGridColumn</i> as a parameter may see this exception raised.
E_CONTROL_NOT_RESPONDING	The control is not responding. Raised after checking whether a specified action took place.
E_COORD_OFF_SCREEN	The specified mouse coordinate is off the screen.
E_COORD_OUTSIDE_WINDOW	The specified coordinate is outside the window. This exception is never raised if the <i>OPT_VERIFY_COORD</i> option is set to <i>FALSE</i> .
E_CURSOR_TIMEOUT	The cursor timeout period was reached before the correct cursor appeared.
E_DELAY_INVALID	The specified delay is not valid.
E_FUNCTION_NOT_REGISTERED	The function called is a user-defined function that hasn't been registered by the application.
E_GRID_HAS_NO_COL_HDR	The specified <i>DataGrid</i> has no column header. All <i>DataGrid</i> methods that use <i>DataGridCell</i> , <i>DataGridRow</i> , or <i>DataGridColumn</i> as a parameter may see this exception raised.
E_GUIFUNC_ID_INVALID	The specified function is not a valid function.
E_INTERNAL	Internal Silk Test Classic error.
E_INVALID_REQUEST	Invalid argument count or argument, or wrong number of arguments.
E_ITEM_INDEX_INVALID	The specified index is not a valid item index.
E_ITEM_NOT_FOUND	The specified item was not found.
E_ITEM_NOT_VISIBLE	The specified item is not visible.
E_KEY_COUNT_INVALID	The repeat count used in the key specification is not a valid number.
E_KEY_NAME_INVALID	The specified key name is not valid.
E_KEY_SYNTAX_ERROR	The syntax used in the key specification is not valid.
E_LINE_COUNT_INVALID	The specified line count is not valid.
E_LINE_NUM_INVALID	The specified line number is not valid.

Exception value	Description
E_LINE_START_EXCEEDS_END	The specified start line exceeds the end line number.
E_MOUSE_BUTTON_INVALID	The specified mouse button is not valid
E_NO_ACTIVE_WINDOW	No window is active.
E_NO_COLUMN	GuptaTable exception.
E_NO_DEFAULT_PUSHBUTTON	The dialog box does not have a default button.
E_NO_FOCUS_WINDOW	No window has the input focus.
E_NO_SETFOCUS_CELL	GuptaTable exception.
E_NO_SETFOCUS_COLUMN	GuptaTable exception.
E_NO_SETTEXT_CELL	GuptaTable exception.
E_NOFOCUS_CELL	No cell in the Gupta table has input focus.
E_NOFOCUS_COLUMN	No column in the Gupta table has input focus.
E_NOFOCUS_ROW	No row in the Gupta table has input focus.
E_NOT_A_TABLEWINDOW	The specified window is not a Gupta table.
E_OPTION_CLASS_MAP_INVALID	The mapping specified with the <i>OPT_CLASS_MAP</i> option is not valid.
E_OPTION_EVTSTR_LENGTH	The length of the event string given in <i>OPT_MENU_INVOKE_POPUP</i> was too long.
E_OPTION_NAME_INVALID	The specified agent option does not exist.
E_OPTION_TOO_MANY_TAGS	The maximum number of tags was exceeded when specifying buttons and menu items using one or more of these options: <ul style="list-style-type: none"> • <i>OPT_CLOSE_CONFIRM_BUTTONS</i> • <i>OPT_CLOSE_WINDOW_BUTTONS</i> • <i>OPT_CLOSE_WINDOW_MENUS</i>
E_OPTION_TYPE_MISMATCH	Mismatch between type of agent option and type of specified value.
E_OPTION_VALUE_INVALID	The specified agent option is not valid.
E_OUT_OF_MEMORY	The system has run out of memory.
E_POS_INVALID	The specified position is not valid.
E_POS_NOT_REACHABLE	The specified position cannot be reached. It is out of range of the object.
E_RESOURCE_NOT_FOUND	The widget does not contain the specified resource.
E_ROW_INDEX_INVALID	The specified index is not a valid row index. All <i>DataGrid</i> methods that use <i>DataGridCell</i> , <i>DataGridRow</i> , or <i>DataGridColumn</i> as a parameter may see this exception raised.
E_SBAR_HAS_NO_THUMB	The scroll bar thumb can not be clicked to scroll a page because the scroll bar does not have a thumb.
E_SQLW_BAD_COLUMN_NAME	A bad column name was specified for the Gupta table.

Exception value	Description
E_SQLW_BAD_COLUMN_NUMBER	A bad column number was specified for the Gupta table.
E_SQLW_BAD_ROW_NUMBER	A bad row number was specified for the Gupta table.
E_SQLW_CANT_ENTER_TEXT	GuptaTable exception.
E_SQLW_INCORRECT_LIST	GuptaTable exception.
E_SQLW_NO_EDIT_WINDOW	GuptaTable exception.
E_SQLW_TABLE_WINDOW_HIDDEN	GuptaTable exception.
E_SQLW_TOO_BIG_LIST	GuptaTable exception.
E_SYSTEM	A system operation has failed.
E_TAG_SYNTAX_ERROR	The tag syntax is not valid: invalid coordinate or index, multiple indices specified, the window part is not the last part of the tag, or the tilde (~) is not followed by a child window.
E_TIMER	The specified timer operation is redundant. For example, a pause operation specified for a stopped timer.
E_TRAP_NOT_SET	Attempted to clear a trap that was not set.
E_UNSUPPORTED	The specified method is not supported on the current platform.
E_VAR_EXPECTED	A function or method call has not passed a variable for a required parameter or an expression failed to specify a variable required by an operator.
E_VERIFY	User-specified verification failed.
E_WINDOW_INDEX_INVALID	The tag uses an invalid index number.
E_WINDOW_NOT_ACTIVE	The specified window is not active.
E_WINDOW_NOT_ENABLED	The specified window is not enabled.
E_WINDOW_NOT_EXPOSED	The specified window is not exposed.
E_WINDOW_NOT_FOUND	The specified window is not found. Raised by any method that operates on a window, except <code>Exists</code> .
E_WINDOW_NOT_UNIQUE	The specified identifier does not represent a unique window. Raised by any method that operates on a window. Affected by the value set with the <code>OPT_VERIFY_UNIQUE</code> option. If you receive this exception, you might try using a slightly modified tag syntax to refer to a window with a non-unique tag. You can either include an index number after the object, as in <code>Dbox ("Cancel[2] ")</code> , or you can specify the window by including the text of a child that uniquely identifies the window, such as <code>Dbox/uniqueText/ . . .</code> , where the unique text is the tag of a child of that window.
E_WINDOW_SIZE_INVALID	The window size is too big for the screen or it is negative.

Exception value	Description
E_WINDOW_TYPE_MISMATCH	The specified window is not valid for this method. Raised when the type of window used is not the type the method accepts.

Troubleshooting Java Applications

This section provides solutions for common reasons that might lead to a failure of the test of your standalone Java application or applet. If these do not solve the specific problem that you are having, you can enable your extension manually.

The test of your standalone Java application or applet may fail if the application or applet was not ready to test, the Java plug-in was not enabled properly, if there is a Java recognition issue, or if the Java applet does not contain any Java controls within the **JavaMainWin**.

What Can I Do If the Silk Test Java File Is Not Included in a Plug-In?

If the `SilkTest_Java3.jar` file is not included in the `lib/ext` directory of the plug-in that you are using:

1. Locate the `lib/ext` directory of the plug-in that you are using and check if the `SilkTest_Java3.jar` file is included in this folder.
2. If the `SilkTest_Java3.jar` file is not included in the folder, copy the file from the `javaex` folder of the Silk Test installation directory into the `lib\ext` directory of the plug-in.

What Can I Do If Java Controls In an Applet Are Not Recognized?

Silk Test Classic cannot recognize any Java children within an applet if your applet contains only custom classes, which are Java classes that are not recognized by default, for example a frame containing only an image. For information about mapping custom classes to standard classes, see *Mapping Custom Classes to Standard Classes*. Additionally, you have to set the Java security privileges that are required by Silk Test Classic.

Multiple Machines Testing

This section provides help and troubleshooting information for testing on multiple machines.

Setting Up the Recovery System for Multiple Local Applications

Problem

By default, the recovery system will only work for the single application assigned to the `const wMainWindow`. With distributed testing, you can get recovery on multiple applications by using `multitestcase` instead of `testcase`.

You might ask whether you can get the recovery system to work on multiple applications that are running locally using `multitestcase` locally. The answer is no; `multitestcase` is for distributed testing only.

But you can use the following solution instead, using `testcase`.

Solution

To get recovery for multiple local applications, set up your frame file to do the following:

1. Get standard `wMainWindow` declarations for each application. The easiest way is to select **File > New > Test Frame** for each application, then combine the `wMainWindow` declarations into a single frame file or include them with `use`.
2. Make the global `wMainWindow` a variable of type `WINDOW`, rather than a constant.
3. Assign one of the windows to `wMainWindow` as a starting point.
4. Create a `LIST OF WINDOW` and assign the `wMainWindow` identifier for each application you are dealing with to it.
5. Define a `TestcaseEnter` function so that you reassign the `wMainWindow` variable and call `SetAppState` on each `MainWin` in turn.
6. Define a `TestcaseExit` function so that you reassign the `wMainWindow` variable and call `SetBaseState` on each `MainWin` in turn.
7. Then use `DefaultBaseState`, or your own base state if you want, with each of your test cases. In your test case, use `SetActive` each time you switch from one application to the other.

Example

The example consists of two sample files. The sample files are for the Classic Agent. If you want to use the example with the Open Agent, you have to change the sample code. For the sample script file, see `two_apps.t`. For the sample include file, see `two_apps.inc`. The example uses two demo applications shipped with Silk Test Classic, the Text Editor and the Test Application. To see that the recovery system is working for both applications, turn on the two debugging options in **Runtime Options** and look at the transcript after running the test script.

The first test case has an intentional error in its last statement to demonstrate the recovery system. The test case also demonstrates how to move data from one application to another with `Clipboard.GetText` and `Clipboard.SetText`.

Because the recovery system is on, the `DefaultBaseState` will take care of invoking each application if it is not already running and will return to the `DefaultBaseState` after each test case, even if the test case fails.

You can print the sample files out or copy them to the **Clipboard**, then paste them into Silk Test Classic. You might have to do some cleanup where the indentation of lines is incorrect in the pasted file.

two_apps.t

The following sample script file for the Classic Agent shows how you can locally test multiple applications. To use the sample with the Open Agent, you have to change the sample code, for example you have to replace all tags with locators.

```
testcase Test1 () appstate DefaultBaseState
  //SetActive each time you switch apps
  TestApplication.SetActive()
  TestApplication.File.New.Pick ()
  MDIChildWindow1.TextField1.SetPosition (1, 1)
  MDIChildWindow1.TextField1.TypeKeys ("In Test Application MDI Child Window
#1.")
  //SetActive each time you switch apps
  TextEditor.SetActive ()
  TextEditor.File.New.Pick ()
  TextEditor.ChildWin("(untitled)[1]").TextField("#1")
```

```

        .TypeKeys ("In Text Editor untitled Document window.<Enter>")
//SetActive each time you switch apps
TestApplication.SetActive()
LIST OF STRING lsTempStrings
lsTempStrings = MDIChildWindow1.TextField1.GetMultiText()
Clipboard.SetText([LIST OF STRING]lsTempStrings)
//SetActive each time you switch apps
TextEditor.SetActive()
TextEditor.ChildWin("(untitled)
[1]").TextField("#1").SetMultiText(Clipboard.GetText(),2)
TextEditor.VerifyCaption("FooBar")

testcase Test2 () appstate DefaultBaseState
wMainWindow = TestApplication
TestApplication.SetActive()
TestApplication.File.New.Pick ()
MDIChildWindow1.TextField1.SetPosition (1, 1)
MDIChildWindow1.TextField1.TypeKeys ("In Test Application MDI Child Window
#1.")
wMainWindow = TextEditor
TextEditor.SetActive ()
TextEditor.File.New.Pick ()
TextEditor.ChildWin("(untitled)[1]").TextField("#1")
        .TypeKeys ("In Text Editor untitled Document window.<Enter>")
wMainWindow = TestApplication
TestApplication.SetActive()
LIST OF STRING lsTempStrings
lsTempStrings = MDIChildWindow1.TextField1.GetMultiText()
Clipboard.SetText([LIST OF STRING]lsTempStrings)
wMainWindow = TextEditor
TextEditor.SetActive()
TextEditor.ChildWin("(untitled)
[1]").TextField("#1").SetMultiText(Clipboard.GetText(),2)

```

two_apps.inc

The following sample include file for the Classic Agent shows how you can locally test multiple applications. To use the sample with the Open Agent, you have to change the sample code, for example you have to replace all tags with locators.

```

// two_apps.inc
// define wMainWindow as a window global var
// and assign one of the apps (your pick) as a starting point.
window wMainWindow = TextEditor
const wMainWindow = TextEditor //replace default def

// Create a list of app MainWins
list of window lwApps = {...}
TextEditor
TestApplication
// Define your own TestCaseEnter.
TestCaseEnter ()
    window wCurrentApp
    for each wCurrentApp in lwApps
        wMainWindow = wCurrentApp
        SetAppState()

// Define your own TestCaseExit.
TestCaseExit (BOOLEAN bException)
    if bException
        ExceptLog()
    window wCurrentApp
    for each wCurrentApp in lwApps

```



```

    wMainWindow = wCurrentApp
    if (wCurrentApp.Exists())
        SetBaseState()

window MainWin TextEditor
    tag "Text Editor"

// The working directory of the application when it is invoked
const sDir = "C:\QAP40"
// The command line used to invoke the application
const sCmdLine = "C:\PROGRAMFILES\\SILKTEST
\TEXTEDIT.EXE"

// The first window to appear when the application is invoked
// const wStartup = ?

// The list of windows the recovery system is to leave open
// const lwLeaveOpen = {?}
Menu File
    tag "File"
    MenuItem New
        tag "New"
    MenuItem Open
        tag "Open"
    MenuItem Close
        tag "Close"
    MenuItem Save
        tag "Save"
    MenuItem SaveAs
        tag "Save As"
    MenuItem Print
        tag "Print"
    MenuItem PrinterSetup
        tag "Printer Setup"
    MenuItem Exit
        tag "Exit"
    Menu Edit
        tag "Edit"
    MenuItem Undo
        tag "Undo"
    MenuItem Cut
        tag "Cut"
    MenuItem Copy
        tag "Copy"
    MenuItem Paste
        tag "Paste"
    MenuItem Delete
        tag "Delete"
    Menu Search
        tag "Search"
    MenuItem Find
        tag "Find"
    MenuItem FindNext
        tag "Find Next"
    MenuItem Replace
        tag "Replace"
    MenuItem GotoLine
        tag "Goto Line"
    Menu Options
        tag "Options"
    MenuItem Font
        tag "Font"
    MenuItem Tabs
        tag "Tabs"

```

```

MenuItem AutomaticIndent
    tag "Automatic indent"
MenuItem CreateBackups
    tag "Create backups"
Menu xWindow
    tag "Window"
MenuItem TileVertically
    tag "Tile Vertically"
MenuItem TileHorizontally
    tag "Tile Horizontally"
MenuItem Cascade
    tag "Cascade"
MenuItem ArrangeIcons
    tag "Arrange Icons"
MenuItem CloseAll
    tag "Close All"
MenuItem Next
    tag "Next"
Menu Help
    tag "Help"
MenuItem About
    tag "About"

window MessageBoxClass MessageBox
    tag "~ActiveApp/[DialogBox]$MessageBox"
    PushButton OK
        tag "OK"
    PushButton Cancel
        tag "Cancel"
    PushButton Yes
        tag "Yes"
    PushButton No
        tag "No"
    StaticText Message
        mswnt tag "#2"
        tag "#1"

window ChildWin Untitled
    tag "(untitled)"
    parent TextEditor
    TextField TextField1
        tag "#1"

window DialogBox Open
    tag "Open"
    parent TextEditor
    StaticText FileNameText
        tag "File Name:"
    TextField FileName1
        tag "File Name:"
    ListBox FileName2
        tag "File Name:"
    StaticText DirectoriesText
        tag "Directories:"
    StaticText CQap40Text
        tag "c:\qap40"
    ListBox CQap40
        tag "c:\qap40"
    StaticText ListFilesOfTypeText
        tag "List Files of Type:"
    PopupList ListFilesOfType
        tag "List Files of Type:"
    StaticText DrivesText
        tag "Drives:"

```

```

PopupList Drives
    tag "Drives:"
PushButton OK
    tag "OK"
PushButton Cancel
    tag "Cancel"
PushButton Network
    tag "Network"

window MainWin TestApplication
    tag "Test Application"
// The working directory of the application when it is invoked
const sDir = "C:\QAP40"

// The command line used to invoke the application
const sCmdLine = "C:\QAP40\TESTAPP.EXE"

// The first window to appear when the application is invoked
// const wStartup = ?

// The list of windows the recovery system is to leave open
// const lwLeaveOpen = {?}
Menu File
    tag "File"
MenuItem New
    tag "New"
MenuItem Close
    tag "Close"
MenuItem Exit
    tag "Exit"
MenuItem About
    tag "About"
Menu Control
    tag "Control"
MenuItem CheckBox
    tag "Check box"
MenuItem ComboBox
    tag "Combo box"
MenuItem ListBox
    tag "List box"
MenuItem PopupList
    tag "Popup list"
MenuItem PushButton
    tag "Push button"
MenuItem RadioButton
    tag "Radio button"
MenuItem StaticText
    tag "Static text"
MenuItem Scrollbar
    tag "Scrollbar"
MenuItem Textfield
    tag "Textfield"
MenuItem DrawingArea
    tag "Drawing area"
MenuItem KeyboardEvents
    tag "Keyboard events"
MenuItem Cursors
    tag "Cursors"
MenuItem ListView
    tag "List view"
MenuItem PageList
    tag "Page list"
MenuItem StatusBar
    tag "Status bar"

```

```
MenuItem ToolBar
  tag "Tool bar"
MenuItem TrackBar
  tag "Track bar"
MenuItem TreeView
  tag "Tree view"
MenuItem UpDown
  tag "Up-Down"
Menu Menu
  tag "Menu"
MenuItem TheItem
  tag "The item"
MenuItem TheAcceleratorItem
  tag "The accelerator item"
Menu TheCascadeItem
  tag "The cascade item"
  MenuItem Item1
    tag "Item1"
  MenuItem Item2
    tag "Item2"
MenuItem Check
  tag "Check"
MenuItem Uncheck
  tag "Uncheck"
MenuItem TheCheckItem
  tag "The check item"
MenuItem Enable
  tag "Enable"
MenuItem Disable
  tag "Disable"
MenuItem TheEnableItem
  tag "The enable item"
Menu Submenu1
  tag "Submenu1"
  MenuItem Item1
    tag "Item1"
  MenuItem Item2
    tag "Item2"
Menu Submenu2
  tag "Submenu2"
  MenuItem Item1
    tag "Item1"
  MenuItem Item2
    tag "Item2"
Menu Submenu3
  tag "Submenu3"
  MenuItem Item1
    tag "Item1"
  MenuItem Item2
    tag "Item2"
MenuItem ThePopupMenu
  tag "The popup menu"
MenuItem Check
  tag "Check"
MenuItem Uncheck
  tag "Uncheck"
MenuItem TheCheckItem
  tag "The check item"
MenuItem Enable
  tag "Enable"
MenuItem Disable
  tag "Disable"
MenuItem TheEnableItem
  tag "The enable item"
```

```

MenuItem AddMenu
  tag "Add menu"
MenuItem ClearMenus
  tag "Clear menus"
Menu DisabledMenu
  tag "DisabledMenu"
MenuItem Item1
  tag "Item1"
MenuItem Item2
  tag "Item2"
Menu Menu5
  tag "#5"
MenuItem MenuItem1
  tag "#1"
MenuItem MenuItem2
  tag "#2"
Menu xWindow
  tag "Window"
MenuItem Cascade
  tag "Cascade"
MenuItem Tile
  tag "Tile"
MenuItem ArrangeIcons
  tag "Arrange Icons"
MenuItem CloseAll
  tag "Close All"
MenuItem ChangeCaption
  tag "Change Caption"
MenuItem SysModall
  tag "SysModal 1"
MenuItem SysModal2
  tag "SysModal 2"
MenuItem SysModal3
  tag "SysModal 3"
MenuItem N1MDIChildWindow1
  tag "1 MDI Child Window #1"

window ChildWin MDIChildWindow1
  tag "MDI Child Window #1"
  parent TestApplication
  TextField TextField1
    tag "#1"

```

Other Problems

This section provides help and troubleshooting information for problems that are not covered by another section.

Adding a Property to the Recorder

1. Write a method.
2. Add a property to the class.
3. Add the property to the list of property names.

For example, if you have a text field that is `ReadOnly` and you want to add that property to the recorder you can do the following:

1. Write the method `Boolean IsReadOnly()` for the `TextField` class.
2. Add the property, `bReadOnly` to the class.

3. Add `bReadOnly` to the list of property names.
4. Compile. `bReadOnly` will appear in the **Recorder** after you compile.

```
Winclass TextField : TextFieldBOOLEAN IsReadOnly()  
STRING sOriginalText = this.GetText()  
STRING sNewText = "xxx"  
this.SetText(sNewText)  
if this.GetText()==sOriginalText  
return TRUE  
else  
return FALSE  
property bReadOnly  
BOOLEAN Get()  
return this.IsReadOnly()  
LIST OF STRING IsPropertyNames = {...}  
"bReadOnly"
```

Cannot Double-Click a Silk Test Classic File and Open Silk Test Classic

Problem

Silk Test Classic does not open automatically when you double-click a `.t`, `.inc`, `.s`, `.g.t`, `.pln`, `.res`, `.stp`, or `.vtp` file.

Cause

During the install process, Silk Test Classic is associated with these file types. However if these file type associations have been changed after Silk Test Classic setup, these file types may not be opened with Silk Test Classic when double-clicking such a file.



Note: File type associations are only available for Microsoft Windows platforms.

Solution

You can either manually associate these file types with Silk Test Classic in Windows, under **Start > Settings > Control Panel > Folder Options**, or reinstall Silk Test Classic.

Cannot Extend AnyWin, Control, or MoveableWin Classes

The `AnyWin`, `Control`, and `MoveableWin` classes are logical (virtual) classes that do not correspond to any actual GUI objects, but instead define methods common to the classes that derive from them. This means that Silk Test Classic never records a declaration that has one of these classes.

Furthermore, you cannot extend or override logical classes. If you try to extend a logical class, by adding a method, property or data member to it, that method, property, or data member is not inherited by classes derived from the class. You will get a compilation error saying that the method, property, or data member is not defined for the window that tries to call it.

You can also not override the class, by rewriting existing methods, properties, or data members. Your modifications are not inherited by classes derived from the class.

Cannot open results file

Problem

Silk Test Classic crashes while running a script and reports the error `Can't open results file`.

Solution

While Silk Test Classic is running a script, it temporarily stores results in a journal file (`.jou`) which is converted to a `.res` file when the script finishes running.

To solve this issue, delete all `.jou` files in the same directory as the script. You do not have to delete your results files.

Then restart Silk Test Classic and run your script again.

Common DLL Problems

Here are some issues that could come up if you are calling DLL functions in a script.

Difficulty creating DLLs to use with Silk Test

Only specific data types are compatible with 4Test. These data types are listed in *C data types for DLL functions*.

If your DLL calls have data types not supported by 4Test, then the functions must be wrapped such that only compatible data types are used for the return type and arguments of the function. Any data types can be used inside the DLL function.

Error after compile: dll not found

In the DLL declaration, use the fully qualified path of the DLL, not just the file name.

Error executing 'CallDllFunction'

When the communication timeout between the Open Agent and the application under test is too small, the `CallDllFunction` might generate the error message `Error executing 'CallDllFunction'`.
`Communication timeout between agent and application.`

To solve this issue, increase the timeout.

1. Stop the Open Agent.
2. Open the file `%OPEN_AGENT_HOME%/agent/openAgent.ini`.
3. Append the following code to the file:

```
-DDefaultCommTimeout=300000
```

This code sets the timeout to 300000 milliseconds, which means 5 minutes. To specify no timeout, set the value to `-1`.

4. Restart the Open Agent.

Error in results file: dll could not be loaded

Make sure the directory containing the DLL is on the path.

Error in results file: dll not found

This usually means that your path does not include the directory containing the DLL. If you are running remotely, make sure that the path on the machine running the agent includes the DLL directory.

Error in results file: function <name> not found in dll

The most likely scenario is that the DLL is a C++ library and the function name has been mangled. To use functions in a C++ library, you need to wrap the functions with the C wrapper and recompile. Then Silk Test Classic can access the function in the library.

If this is not the problem, there might be a typo in the function name in the DLL.

Warning in results file: String buffer size was increased from x to 256 characters

If the user calls a DLL function with an output string buffer that is less than the minimum size of 256 characters, the original string buffer is resized to 256 characters and a warning is printed. This warning, *String buffer size was increased from x to 256 characters* (where x is the length of the given string plus one) alerts the user to a potential problem where the buffer used might be shorter than necessary.

Common Scripting Problems

Here are some common problems that occur with scripts.

Typographical errors

It is very easy to make typographical errors that the 4Test compiler cannot catch. If a line of code does nothing, this might be the problem.

Global variables with unexpected values

When you write a function that uses global variables, make sure that each variable has an appropriate value when the function exits. If another function uses the same variable later, and it has an unexpected value on entry to the function, an error could occur.

To check that a variable has a reasonable value on entry to a function, set a breakpoint on the line that calls the function and use the command **View > Global Variables** to check the variable's value.

Uninitialized variables

Silk Test Classic does not initialize variables for you. So if you have not initialized a variable on entry to a function, it will have the value `<unset>`. It is better to explicitly give a value to a variable than to trust that another function has already initialized it for you. Also, remember that 4Test does not keep local variables around after a function exits; the next time the function is called, its local variables could be uninitialized.

If you are in doubt about whether a variable has a reasonable value at a particular point, set a breakpoint there and use **View > Global Variables** or **View > Local Variables** to check the variable's value.

Global and local variables with the same name

It is usually not good programming practice to give different variables the same names. If a global and local variable with the same name are in scope (accessible) at the same time, your code can only access the local variable.

To check for repeated names, use **View > Local Variables** and **View > Global Variables** to see if two variables with the same name are in scope simultaneously.

Incorrect values for loop variables

When you write a for loop or a while loop, be sure that the initial, final, and step values for the variable that controls the loop are correct. Incrementing a loop variable one time more or less than you really want is a common source of errors.

To make sure a control loop works as you expect, use **Debug > Step Into** to step through the execution of the loop one statement at a time, and watch how the value of the loop variable changes using **View > Local Variables**.

Checking the precedence of operators

The order in which 4Test applies operators when it evaluates an expression may not be what you expect. Use parentheses, or break an expression down into intermediate steps, to make sure it works as expected. You can use **View/Expression** to evaluate an expression and check the result.

Incorrect uses of break statements

A break statement transfers control of the script out of the innermost nested for, for each, while, switch, or select statement only. In other words, break exits from a single loop level, not from multiple levels. Use **Debug > Step Into** to step through the script one line at a time and ensure that the flow of control works as you expect.

Infinite loops

To check for infinite loops, step through the script with **Debug > Step Into**.

Code that never executes

To check for code that never executes, step through the script with **Debug > Step Into**.

Conflict with Virus Detectors

Problem

Silk Test Classic will occasionally have problems on machines running virus detectors that use heuristic or algorithmic virus detection in addition to the standard pattern recognition. What happens is that while Silk Test Classic is running, the virus detector identifies Silk Test Classic as displaying "virus-like" behavior, and kills or otherwise disables the agent. This leads to unpredictable and inconsistent behavior in Silk Test Classic, including loss of communications with the agent and inconsistent test results or object recognition.

Solution

To avoid this problem the only solution is to temporarily disable the virus detector while Silk Test Classic is running.

Displaying the Euro Symbol

Problem

You want to display the Euro (€) symbol.

Solution

Download a Euro-enabled font from Microsoft. Double check that you can see the Euro symbol by opening Notepad on the machine where you installed the font and entering the ASCII code for the Euro symbol. As long as you see the symbol in notepad, you should be able to see it within Silk Test Classic.

In Silk Test Classic, click **Options > Editor Font** and be sure that your font is set to Arial, Courier New, or Times New Roman.

Do I Need Administrator Privileges to Run Silk Test Classic?

You require the following privileges to install or run Silk Test Classic:

- To install Silk Test Classic, you must have local administrator privileges.
- To install Silk Test Classic on a Windows server, you must have domain-level administrator privileges.
- To run Silk Test Classic with the Classic Agent, you must have administrator privileges.
- If you have installed Silk Test Classic into the `Program Files` folder, you must have administrator privileges to run Silk Test Classic with the Open Agent.
- If you have installed Silk Test Classic into a different location than the `Program Files` folder, you do not need to have administrator privileges to run Silk Test Classic with the Open Agent.
- To run Silk Test Classic, you require full access rights to the following folders, including all subfolders:
 - `C:\ProgramData\Silk\SilkTest.`
 - `%APPDATA%\Roaming\Silk\SilkTest.`
 - `%APPDATA%\Local\Silk\SilkTest.`
 - `%TEMP%.`



Note: If User Account Control (UAC) is activated on your system, Micro Focus recommends that you install Silk Test Classic into a different location than the `Program Files` folder.

General Protection Faults

Problem

When recording or running tests, you get a `General Protection Fault (GPF)` or `Invalid Page Fault (IPF)` in `agent.exe` or `partner.exe`.

Solution

It can be very difficult to pin down the cause of these problems. It might involve a combination of your machine's configuration, other applications that are running, and the network's configuration. The best approach is to gather the diagnostic information described below and send it to Technical Support with a detailed description of what scenario led to the error.

Capture the system diagnostics

When the system error message displays, chose the option to capture detailed information on the error. Write the information down.

Capture a debug.log file

1. Ensure that no Silk Test Classic or Agent processes are running.
2. Open a DOS prompt window.
3. Change your working directory to your Silk Test Classic installation directory.
4. Delete or rename `c:\debug.log` if the file exists.
5. Set the following environment variable: `set QAP_DEBUG_AGENT=1.`
6. Start the Agent manually: `start .\agent.`
7. Start Silk Test Classic manually: `start .\partner.`
8. Go through the scenario to reproduce the problem.
9. The file `c:\debug.log` file will be created.
10. Send this file as an attachment to your email to Technical Support.

Monitor CPU and RAM usage	When reproducing this error to gather the diagnostics above, also run a system resource monitor to check on CPU and RAM usage. Note whether CPU or RAM is being exhausted.
Note your system configuration	When sending in these diagnostics, note the version of Silk Test Classic, the operating system and version, and the machine configuration (CPU, RAM, disk space).

Running Global Variables from a Test Plan Versus Running Them from a Script

Problem

When running from a test plan, global variables don't keep their value from one test case to another.

When test cases are run from a script, global variables are initialized once at the beginning and do not get reset while the script is being run. On the other hand, when you run test cases from a test plan, all global variables get re-initialized after each test case. This is because the Agent reinitializes itself before running each test case. Consequently, you may find that global variables are not as useful when running from a test plan.

Solution

A workaround is to use the `FileWriteLine` or `FileWriteValue` function to write the values of the global variables out to a file, then use the `FileReadLine` or `FileReadValue` function to read the value back into each variable in each test case.

Include File or Script Compiles but Changes are Not Picked Up

Problem

You compile an include file or script, but changes that you made are not used when you run the script.

Solutions

Did you change the wrong include file? Make sure that the include file you are compiling is the same as the file that is being used by the script. Just because you have an include file open and have just compiled it does not mean that it is being used by the script. The include file that the script will use is either specified in Runtime Options (Use Files field) or by a use statement in the script.

Is there a time-stamp problem? If the time stamp for the file on disk is later than the machine time when you do **Run > Compile**, then the compile does not actually happen and no message is given. This can happen if two machines are sharing a file system where the files are being written out and the time on the machines is not synchronized.

By default, Silk Test Classic only compiles files that need compiling, based on the date of the existing object files and the system clock. This way, you don't have to wait to recompile all files each time a change is made to one file.

If you need to, you can force Silk Test Classic to compile all files by selecting **Run > Compile All**. **Run > Compile All** compiles the script or suite and all dependent include files, even if they have not changed since they were last compiled. It also

compiles files listed in the **Use Files** field in the **Runtime Options** dialog and the compiler constants declared in the **Runtime Options** dialog. Finally, it compiles the include files loaded at startup, if needed.

Are your object files corrupted? Sometimes a Silk Test Classic object (.ino or .to) file can become corrupted. Sometimes a corrupted object file can cause Silk Test Classic to assume that the existing compile is up to date and to skip the recompile without any message.

To work around this, delete all .ino and .to files in the directories containing the .inc and .t files you are trying to compile, then compile again.

Library Browser Not Displaying User-Defined Methods

Problem

You add a description for a user-defined method and a user-defined function to `4test.txt`. After restarting Silk Test Classic, the new description for the function displays in the Library Browser, but not the description for the method. So you know that the modified `4test.txt` file is being used, but your user-defined method is not being displayed in the **Library Browser**.

Solutions

Only methods defined in a class definition (that is, in your include file where your class is defined) will display in the **Library Browser**. For example, `MyAccept` will be displayed.

```
winclass DialogBox:DialogBox
Boolean MyAccept()
...
```

Methods you define for an individual object are not displayed in the **Library Browser**. For example, `MyDialogAccept` will not display.

```
DialogBox MyDialog
tag "My Dialog"
Boolean MyDialogAccept()
...
```

In order to display in the **Library Browser**, the description in your `4test.txt` file must have a return type that matches the return type in your include file declaration. If the `4test.txt` description has no `returns` statement, then the declaration must be for a return type of `void` (either specified explicitly or by defaulting to `void`). Otherwise, the description will not display in the **Library Browser**.

For more information about adding information to the **Library Browser**, see *Adding to the Library Browser*.

Maximum Size of Silk Test Classic Files

The following size limits apply:

- The limit for .inc, .t, and .pln files (and their associated backup files, .*_) is 64K lines.
- The size limit for the corresponding object files (. *o) depends on the amount of available system memory.
- The Silk Test Classic editor limits lines to 1024 characters.
- The maximum size of a single entry in a .res file is 64K.
- Test case names can have a maximum of 127 characters. When you create a data-driven test case, Silk Test Classic truncates any test case name that is greater than 124 characters.

Recorder Does Not Capture All Actions

Problem

While recording, the Silk Test Recorder does not capture all actions in your application under test, though you complete the actions.

Cause

The application under test may be "going too fast" and the Silk Test Recorder may not be able to keep up.

Solution

Slow down the interactions with your application while recording. Record a test case at the speed of the Silk Test Recorder.

Relationship between Exceptions Defined in 4test.inc and Messages Sent To the Result File

Silk Test Classic calls `LogError` automatically when it raises an exception that you have not handled. By reading `4test.inc` you can find that Silk Test Classic has a list of exceptions like:

```
E_ABORT = -10100,  
E_TBL_HAS_NO_ROW_HDR = -30100,  
E_WINDOW_NOT_FOUND = -27800
```

Since exception numbers can apply to more than one exception, it can be helpful to query on a particular exception number via `ExceptNum()` to decide how to handle an error. If you need to query on a specific exception message, you can use `ExceptData()`. We recommend using `MatchStr()` with `ExceptData()`.

To find the `E_...` constant for any 4Test exception, you can use:

```
[ - ] do  
    <code that causes exception>  
[ - ] except  
[ ] LogWarning ("Exception number: {[EXCEPTION]ExceptNum ()}")  
[ ] reraise
```

This will print out the exception constant in the warning.

Be sure to remove the `LogWarning do...except` block after you have found the `E_...` constant.

The 4Test Editor Does Not Display Enough Characters

Problem

While you can edit 4Test files outside of Silk Test Classic and create lines with more than 1024 characters, the Silk Test 4Test Editor (4Test Editor) does not let you edit or extend these lines.

The line limit of the 4Test Editor is 1024 characters.

Solution

Use the `<Shift+Enter>` continuation character to break the line into smaller lines.

Stopping a Test Plan

Problem

You want to abort a test plan programmatically without using `exit`. Calling `exit` just aborts the script and continues on to the next test case.

Solution

You can call

```
[ ] @("$StopRunning") ()
```

from a test case or a recovery system function such as `ScriptExit()`, which is called for each test case in the test plan, or `TestCaseExit()`.

This call will stop everything without even invoking the recovery system. Calling it will generate the following exception message, with no call stack: `Exception -200000`

Using a Property Instead of a Data Member

Data members are resolved (assigned values) during compilation. If the expression for the data member includes variables that will change at run-time, then you must use a property instead of that data member.

Using File Functions to Add Information to the Beginning of a File

In Silk Test Classic 5.5 SP1 or later, there is no file open mode that allows you to insert information into the beginning of a file. If you use `FM_UPDATE`, you can read in part of your file before writing, but any write function calls will overwrite the rest of the file.

If you are writing strings rather than structured data, you can use `ListRead()` and `ListWrite()` to insert information at the beginning or any other point of a file. Use `ListRead()` to read the contents of the file into a list, insert the new information at the head or any other point of the list, and use `ListWrite()` to write it back out.

```
[-] LIST OF STRING lsNewInfo = {...}
[ ] "*New line one*"
[ ] "*New line two*"
[ ] "*New line three*"
[ ] LIST OF STRING lsFile
[ ] INTEGER i
[ ]
[ ] ListRead (lsFile, "{GetProgramDir ()}\Sample.txt")
[-] for i = 1 to ListCount (lsNewInfo)
[ ] ListInsert (lsFile, i, lsNewInfo[i])
[ ] ListWrite (lsFile, "{GetProgramDir ()}\Sample.txt")
[ ]
```

Sample.txt before the write:

```
Line 1
Line 2
Line 3
Line 4
Line 5
```

Sample.txt after:

```
*New line one*
*New line two*
```

```
*New line three*
Line 1
Line 2
Line 3
Line 4
Line 5
```

Why Does the Str Function Not Round Correctly?

Any decimal/float number has an internal binary representation. Unfortunately, you can never be sure if a decimal value has an exact representation in its binary pendant. If an exact binary representation is not possible (mathematical constraint), the nearest value is used and this leads to the issue where it seems the `str` function is not rounding correctly. You can workaround this issue. Use the following code to see the internal representation:

```
[ ] printf("%.a20e\n", 32.495)
[ ] printf("%.a20e\n", 31.495)
```

Troubleshooting Projects

This section provides solutions to common problems that you might encounter when you are working with projects in Silk Test Classic.

Files Not Found When Opening Project

If, when opening your project, Silk Test Classic cannot find a file in the location referenced in the project file, which is a `.vtp` file, an error message displays noting the file that cannot be found.

Silk Test Classic may not be able to find files that have been moved or renamed outside of Silk Test Classic, for example in Windows Explorer, or files that are located on a shared network folder that is no longer accessible.

- If Silk Test Classic cannot find a file in your project, we suggest that you note the name of missing file, and click **OK**. Silk Test Classic will open the project and remove the file that it cannot find from the project list. You can then add the missing file to your project.
- If Silk Test Classic cannot open multiple files in your project, we suggest you click **Cancel** and determine why the files cannot be found. For example a directory might have been moved. Depending upon the problem, you can determine how to make the files accessible to the project. You may need to add the files from their new location.

Silk Test Classic Cannot Load My Project File

If Silk Test Classic cannot load your project file, the contents of your `.vtp` file might have changed or your `.ini` file might have been moved.

If you remove or incorrectly edit the `ProjectIni=` line in the `ProjectProfile` section of your `<projectname>.vtp` file, or if you have moved your `<projectname>.ini` file and the `ProjectIni=` line no longer points to the correct location of the `.ini` file, Silk Test Classic is not able to load your project.

To avoid this, make sure that the `ProjectProfile` section exists in your `.vtp` file and that the section refers to the correct name and location of your `.ini` file. Additionally, the `<projectname>.ini` file and the `<projectname>.vtp` file refer to each other, so ensure that these references are correct in both files. Perform these changes in a text editor outside of Silk Test Classic.

Example

The following code sample shows a sample `ProjectProfile` section in a `<projectname>.vtp` file:

```
[ProjectProfile]
ProjectIni=C:\Program Files\
\SilkTest\Projects\.ini
```

Silk Test Classic Cannot Save Files to My Project

You cannot add or remove files from a read-only project. If you attempt to make any changes to a read-only project, a message box displays indicating that your changes will not be saved to the project.

For example, Unable to save changes to the current project. The project file has read-only attributes.

When you click **OK** on the error message box, Silk Test Classic adds or removes the file from the project temporarily for that session, but when you close the project, the message box displays again. When you re-open the project, you will see your files have not been added or removed.

Additionally, if you are using Microsoft Windows 7 or later, you might need to run Silk Test Classic as an administrator. To run Silk Test Classic as an administrator, right-click the Silk Test Classic icon in the **Start Menu** and click **Run as administrator**.

Silk Test Classic Does Not Run

The following table describes what you can do if Silk Test Classic does not start.

If Silk Test Classic does not run because it is looking for the following:	You can do the following:
Project files that are moved or corrupted.	Open the <code>SilkTestClassic.ini</code> file in a text editor and remove the <code>CurrentProject=</code> line from the <code>ProjectState</code> section. Silk Test Classic should then start, however your project will not open. You can examine your <code><projectname>.ini</code> and <code><projectname>.vtp</code> files to determine and correct the problem. The following code example shows the <code>ProjectState</code> section in a sample <code>partner.ini</code> file: <pre>[ProjectState] CurrentProject=C:\Program Files \SilkTest install directory \SilkTest\Examples\ProjectName.vtp</pre>
A <code>testplan.ini</code> file that is corrupted.	Delete or rename the corrupted <code>testplan.ini</code> file, and then restart Silk Test Classic.

My Files No Longer Display In the Recent Files List

After you open or create a project, files that you had recently opened outside of the project do no longer display in the **Recent Files** list.

Cannot Find Items In Classic 4Test

If you are working with Classic 4Test, objects display in the correct nodes on the **Global** tab, however when you double-click an object, the file opens and the cursor displays at the top of the file, instead of in the line in which the object is defined.

Editing the Project Files

You require good knowledge of your files and how the partner and `<projectname>.ini` files work before attempting to edit these files. Be cautious when editing the `<projectname>.vtp` and `<projectname>.ini` files.

To edit the `<projectname>.vtp` and `<projectname>.ini` files:

1. Update the references to the source location of your files. If the location of your `projectname.vtp` and `projectname.ini` files has changed, make sure you update that as well. Each file refers to the other.

The `ProjectProfile` section in the `projectname.vtp` file is required. Silk Test Classic will not be able to load your project if this section does not exist.

1. Ensure that your project is closed and that all the files referenced by the project exist.
2. Open the `<projectname>.vtp` and `<projectname>.ini` files in a text editor outside of Silk Test Classic.



Note: Do not edit the `projectname.vtp` and `projectname.ini` files in the 4Test Editor.

3. Update the references to the source location of your files.
4. The `<projectname>.vtp` and `<projectname>.ini` files refer to each other. If the relative location of these files has changed, update the location in the files.

The `ProjectProfile` section in the `<projectname>.vtp` file is required. Silk Test Classic is not able to load your project if this section does not exist.

Recognition Issues

This section provides help and troubleshooting information for recognition issues.

How Can the Application Developers Make Applications Ready for Automated Testing?

The attributes available for a specific control in the application under test (AUT) might not be sufficient to guarantee that Silk Test Classic always recognizes the control during automated testing. In such a case the application developer can add custom attributes to the control, which can then be used as locator attributes for the control. The following examples describe how an application developer can include custom attributes in different application types:

- To include custom attributes in a Web application, add them to the html tag. Type `<input type='button' bcauid='abc' value='click me' />` to add an attribute called `bcauid`.
- To include custom attributes in a Java SWT application, use the `org.swt.widgets.Widget.setData(String <varname>key</varname>, Object <varname>value</varname>)` method.
- To include custom attributes in a Swing application, use the `putClientProperty("propertyName", "propertyValue")` method.

Tips

This section provides general troubleshooting tips.

Example Test Cases for the Find Dialog Box

If you want to test the **Find** dialog box, each test case would need to perform the following tasks:

1. Open a new document file.
2. Type text into the document.
3. Position the insertion point at the top of the file.
4. Select **Find** from the **Search** menu.
5. Select the forward (down) direction for the search.
6. Make the search case sensitive.

Non-Data-Driven Test Case for the Classic Agent

```
testcase FindTest ()
  TextEditor.File.New.Pick ()
  DocumentWindow.Document.TypeKeys ("Test Case<HOME>")
  TextEditor.Search.Find.Pick ()
  Find.FindWhat.SetText ("Case")
  Find.CaseSensitive.Check ()
  Find.Direction.Select ("Down")
  Find.FindNext.Click ()
  Find.Cancel.Click ()
  DocumentWindow.Document.VerifySelText (<text>)
  Case
  TextEditor.File.Close.Pick ()
  MessageBox.No.Click ()
```

The major disadvantage of this kind of test case is that it tests only one out of the many possible sets of input data to the **Find** dialog box. To adequately test the **Find** dialog box, you must record or hand-write a separate test case for each possible combination of input data that needs to be tested. In even a small application, this creates a huge number of test cases, each of which must be maintained as the application changes.

Non-Data-Driven Test Case for the Open Agent

```
testcase Find ()
  recording
    UntitledNotepad.SetActive()
    UntitledNotepad.New.Pick()
    UntitledNotepad.TextField.TypeKeys("Test Case
<LessThan>Home")
    UntitledNotepad.TextField.PressKeys("<Left Shift>")
    UntitledNotepad.TextField.TypeKeys("<GreaterThan>")
    UntitledNotepad.Find.Pick()
    UntitledNotepad.FindDialog.FindWhat.SetText("Case")
    UntitledNotepad.FindDialog.Down.Select("Down")
    Tmp_findNotepad.Find.MatchCase.Check()
    UntitledNotepad.FindDialog.FindNext.Click()
    Tmp_findNotepad.Find.Cancel.Click()
    Tmp_findNotepad.Find.Close()
```

When to use the Bitmap Tool

You might want to use the **Bitmap Tool** in these situations:

- To compare a baseline bitmap against a bitmap generated during testing.
- To compare two bitmaps from a failed test.

For example, suppose during your first round of testing you create a bitmap using one of Silk Test Classic's built-in bitmap functions, `CaptureBitmap`. Assume that a second round of testing generates another bitmap, which your test script compares to the first. If the testcase fails, Silk Test Classic raises an exception but cannot specifically identify the ways in which the two images differ. At this point, you can open the **Bitmap Tool** from the results file to inspect both bitmaps.

Troubleshooting Web Applications

The test of your browser application may have failed for one of the reasons described in this section. If the suggested solutions do not address the problem you are having, you can enable your extension manually.

What Can I Do If the Page I Have Selected Is Empty?

If the page you are testing is empty or does not contain any HTML elements, you might receive a `Could not recognize any HTML classes in your browser application message`. Your configuration might be correct, however, the automated configuration test does not support testing of blank pages or pages that do not contain HTML elements. You can manually verify that your extensions are set properly, open your application, and then record window declarations. If you can record against HTML classes, the extension is configured correctly and you are ready to set up the recovery system using the **Basic Workflow** bar.

Why Do I Get an Error Message When I Set the Accessibility Extension?

If you are using Internet Explorer to test a Web application and you have set the Accessibility extension, you might get an error message when the start page of the browser is "about:blank". To avoid getting the error message, set the start page of the browser to a different page.

Using the Runtime Version of Silk Test Classic

The Silk Test Classic Runtime (Runtime) provides a subset of the functionality of Silk Test Classic. Specifically, it allows you to perform all of the tasks associated with executing tests and analyzing results. You are prohibited from editing existing automation or creating new automation. The Runtime is intended to run previously compiled files. If you update a shared file while the Runtime is open, you must close the Runtime and reopen it in order to use the updated file.


Silk Test Classic Runtime is an installation option. For additional information, refer to the *Silk Test Installation Guide*.

The *Silk Test Classic Runtime Help* includes the topics that are available from the full version of Silk Test Classic, and additional product-specific information.

Installing the Runtime Version

Silk Test Classic Runtime is an installation option. For additional information, refer to the *Silk Test Installation Guide*.

We strongly recommend that you do not install Silk Test Classic Runtime on the same machine as Silk Test Classic. Silk Test Classic runtime shares files with this product and will overwrite any other installation you already have on your machine.

 **Note:** Silk Test Classic Runtime is sold and licensed separately from standard Silk Test Classic.

Starting the Runtime Version

You can start Silk Test Classic Runtime from the following locations:

- The command-line prompt in a DOS window. Enter `runtime.exe`. The same syntax applies as with starting Silk Test Classic from the command line.
- The Silk Test Classic GUI. You must have selected the Silk Test Classic Runtime option during installation.

When you start the Runtime, it displays minimized as an icon only; click the icon to maximize the window.

Comparing Silk Test Classic and Silk Test Classic Runtime Menus and Commands

The table below lists the menus and commands that are available for each agent in Silk Test Classic and those that are available in Silk Test Classic Runtime:

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
Breakpoint	Toggle	Classic Agent	No
		Open Agent	

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	Add	Classic Agent Open Agent	No
	Delete	Classic Agent Open Agent	No
	Delete All	Classic Agent Open Agent	No
Debug	Abort	Classic Agent Open Agent	No
	Exit	Classic Agent Open Agent	No
	Finish Function	Classic Agent Open Agent	No
	Reset	Classic Agent Open Agent	No
	Run and Debug/Continue	Classic Agent Open Agent	No
	Run to Cursor	Classic Agent Open Agent	No
	Step Into	Classic Agent Open Agent	No
	Step Over	Classic Agent Open Agent	No
Edit	Undo	Classic Agent Open Agent	No
	Redo	Classic Agent Open Agent	No
	Cut	Classic Agent Open Agent	No
	Copy	Classic Agent Open Agent	Yes
	Select All	Classic Agent Open Agent	Yes
	Paste	Classic Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Delete	Classic Agent Open Agent	No
	Find	Classic Agent Open Agent	Yes
	Find Next	Classic Agent Open Agent	Yes
	Replace	Classic Agent Open Agent	No
	Go to Line	Classic Agent Open Agent	Yes
	Go to Definition	Classic Agent Open Agent	Yes
	Find Error	Classic Agent Open Agent	Yes
	Data Driven	Classic Agent Open Agent	No
	Visual 4Test	Classic Agent Open Agent	Yes
File	New	Classic Agent Open Agent	No
	Open	Classic Agent Open Agent	Yes
	Close	Classic Agent Open Agent	Yes
	Save	Classic Agent Open Agent	No
	Save Object File	Classic Agent Open Agent	No
	Save As	Classic Agent Open Agent	No
	Save All	Classic Agent Open Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	New Project	Classic Agent Open Agent	No
	Open Project	Classic Agent Open Agent	Yes
	Close Project	Classic Agent Open Agent	Yes
	Export Project	Classic Agent Open Agent	No
	Email Project	Classic Agent Open Agent	No
	Run	Classic Agent Open Agent	Yes
	Debug	Classic Agent Open Agent	No
	Check out	Classic Agent Open Agent	No
	Check in	Classic Agent Open Agent	No
	Print	Classic Agent Open Agent	Yes
	Printer Setup	Classic Agent Open Agent	Yes
	Recent Files and Recent Projects	Classic Agent Open Agent	Yes
	Exit	Classic Agent Open Agent	Yes
Help	Help Topics	Classic Agent Open Agent	Yes
	Library Browser	Classic Agent Open Agent	Yes
	Tutorials	Classic Agent Open Agent	Yes
	About Silk Test Classic	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
Include	Open	Classic Agent Open Agent	Yes
	Open All	Classic Agent Open Agent	Yes
	Close	Classic Agent Open Agent	Yes
	Close All	Classic Agent Open Agent	Yes
	Save	Classic Agent Open Agent	No
	Acquire Lock	Classic Agent Open Agent	No
	Release Lock	Classic Agent Open Agent	No
Options	General	Classic Agent Open Agent	Yes
	Editor Font	Classic Agent Open Agent	Yes
	Editor Colors	Classic Agent Open Agent	Yes
	Runtime	Classic Agent Open Agent	Yes
	Agent	Classic Agent Open Agent	Yes
	Extensions	Classic Agent	Yes
	Application Configurations	Open Agent	Yes
	Recorder	Classic Agent Open Agent	No
	Class Map	Classic Agent	Yes
	Class Attributes	Classic Agent	Yes
	Property Sets	Classic Agent Open Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	TrueLog	Classic Agent Open Agent	Yes
	Silk Central URLs	Classic Agent Open Agent	Yes
	Open Options Set	Classic Agent Open Agent	Yes
	Save New Options Set	Classic Agent Open Agent	No
	Close Options Set	Classic Agent Open Agent	Yes
	Recent Options Sets	Classic Agent Open Agent	Yes
Outline	Move Left	Classic Agent Open Agent	No
	Move Right	Classic Agent Open Agent	No
	Transpose Up	Classic Agent Open Agent	No
	Transpose Down	Classic Agent Open Agent	No
	Expand	Classic Agent Open Agent	Yes
	Expand All	Classic Agent Open Agent	Yes
	Collapse	Classic Agent Open Agent	Yes
	Collapse All	Classic Agent Open Agent	Yes
	Comment	Classic Agent Open Agent	No
	Uncomment	Classic Agent Open Agent	No
Project	View Explorer	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Align	Classic Agent Open Agent	Yes
	Project Description	Classic Agent Open Agent	No
	Add File	Classic Agent Open Agent	No
	Remove File	Classic Agent Open Agent	No
Record	Window Declarations	Classic Agent	No
	Application State	Classic Agent Open Agent	No
	Testcase	Classic Agent Open Agent	No
	Method	Classic Agent Open Agent	No
	Actions	Classic Agent	No
	Class	Classic Agent	No
	Window Identifiers	Classic Agent	No
	Window Locations	Classic Agent Open Agent	No
	Defined Window	Classic Agent	No
	Window Tags	Classic Agent	No
Results	Select	Classic Agent Open Agent	Yes
	Merge	Classic Agent Open Agent	Yes
	Delete	Classic Agent Open Agent	Yes
	Extract	Classic Agent Open Agent	Yes
	Export	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Launch TrueLog Explorer	Classic Agent Open Agent	Yes
	Convert to Plan	Classic Agent Open Agent	No
	Compact	Classic Agent Open Agent	Yes
	Show Summary	Classic Agent Open Agent	Yes
	Hide Summary	Classic Agent Open Agent	Yes
	View Options	Classic Agent Open Agent	Yes
	Go to Source	Classic Agent Open Agent	Yes
	View Differences	Classic Agent Open Agent	Yes
	Update Expected Value	Classic Agent Open Agent	No
	Pass/Fail Report	Classic Agent Open Agent	Yes
	Mark Failures in Plan	Classic Agent Open Agent	Yes
	Compare Two Results	Classic Agent Open Agent	Yes
	Next Result Difference	Classic Agent Open Agent	Yes
	Next Error Difference	Classic Agent Open Agent	Yes
Run	Compile	Classic Agent Open Agent	No
	Compile all	Classic Agent Open Agent	No

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	Run	Classic Agent Open Agent	Yes
	Debug	Classic Agent Open Agent	No
	Application State	Classic Agent Open Agent	Yes
	Testcase	Classic Agent Open Agent	Yes
	Show Status	Classic Agent Open Agent	Yes
	Abort	Classic Agent Open Agent	Yes
Testplan	Go to Script	Classic Agent Open Agent	Yes
	Detail	Classic Agent Open Agent	No
	Insert Template	Classic Agent Open Agent	No
	Completion Report	Classic Agent Open Agent	Yes
	Mark	Classic Agent Open Agent	Yes
	Mark All	Classic Agent Open Agent	Yes
	Unmark	Classic Agent Open Agent	Yes
	Unmark All	Classic Agent Open Agent	Yes
	Mark by Query	Classic Agent Open Agent	Yes
	Mark by Named Query	Classic Agent Open Agent	Yes
	Find Next Mark	Classic Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
		Open Agent	
	Define Attributes	Classic Agent Open Agent	Yes
	Run Manual Tests	Classic Agent Open Agent	No
Tools	Start Silk Performer	Classic Agent Open Agent	No
	Connect to Default Agent	Classic Agent Open Agent	Yes
	Data Drive Testcase	Classic Agent	No
	Enable Extensions	Classic Agent	No
	Open Silk Central	Classic Agent Open Agent	Yes
View/Transcript	Expression	Classic Agent Open Agent	No
	Global Variables	Classic Agent Open Agent	No
	Local Variables	Classic Agent Open Agent	No
	Expand Data	Classic Agent Open Agent	No
	Collapse Data	Classic Agent Open Agent	No
	Module	Classic Agent Open Agent	No
	Breakpoints	Classic Agent Open Agent	No
	Call Stack	Classic Agent Open Agent	No
	Transcript	Classic Agent Open Agent	No
Window	Tile Vertically	Classic Agent Open Agent	Yes

Menu Name	Command	Available in Silk Test Classic	Available in Silk Test Classic Runtime
	Tile Horizontally	Classic Agent Open Agent	Yes
	Cascade	Classic Agent Open Agent	Yes
	Arrange Icons	Classic Agent Open Agent	Yes
	Close All	Classic Agent Open Agent	Yes
	Next	Classic Agent Open Agent	Yes
	Previous	Classic Agent Open Agent	Yes
	# filename-filepath	Classic Agent Open Agent	Yes
Workflows	Basic	Classic Agent Open Agent	No
	Data Driven	Classic Agent Open Agent	No

Working with Files

This section describes how you can use files in Silk Test Classic.

Creating a New File

Use the **New File** dialog box to create a new file.

1. Click **File > New**. The **New File** dialog box appears.

2. Select the file type.

Silk Test Classic supports the following file types:

- Test frame
- Test plan
- 4Test script
- 4Test include file
- Keyword-driven test
- Keyword sequence
- Suite
- Text

3. Click **OK**.

- If you have selected any file type except test frame, a new editing window opens. The title bar of the new editing window displays the file type you have chosen and `untitled`.
- If you have selected to create a new test frame file, and you are using the Open Agent, the **Select Application** dialog box displays, allowing you to create a test frame file for a specific application.
- If you have selected to create a new test frame file, and you are using the Classic Agent, the **New Test Frame** dialog box displays, allowing you to create a test frame file for an open application. For additional information, see *Creating a New Test Frame*.

Searching for a String in a File

Use the **Find** dialog box to find a string in a file.

1. Click **Edit > Find**. The **Find** dialog box appears.

2. Type the text that you want to search for the active window into the **Find what** field.

The default string is the last string you searched for.

3. Check the **Wrap Search** check box to begin and end the search for the specified text at the current location of the cursor.

When the check box is checked, the search continues past the end or beginning of the document back to the current location of the cursor.

4. Check the **Case sensitive** check box to make the search sensitive to the case of the text string.

The default is to search with case sensitivity off.

5. Set the direction for the search.

- Select **Up** to have the search go from the current location of the cursor towards the top, or beginning, of the file.

- Select **Down** to have the search go from the current location of the cursor towards the bottom, or end, of the file. This is the default direction.
6. Click **Find Next** to locate the next occurrence of the text string. If the search is successful, the first occurrence of the text string in the file is highlighted. If the search is not successful, an informational message displays.

Replacing a String in a File

Use the **Replace** dialog box to replace a string in a file with another string.

1. Click **Edit > Replace**. The **Replace** dialog box appears.
2. Type the text that you want to replace into the **Find what** field.
The default string is the last string you searched for.
3. Type the text with which you want to replace the text in the **Find what** field into the **Replace with** field.
4. Check the **Case sensitive** check box to make the search sensitive to the case of the text string.
The default is to search with case sensitivity off.
5. Set the direction for the search.
 - Select **Up** to have the search go from the current location of the cursor towards the top, or beginning, of the file.
 - Select **Down** to have the search go from the current location of the cursor towards the bottom, or end, of the file. This is the default direction.
6. Click **Find Next** to locate the next occurrence of the text in the **Find What** field. If the search is successful, the first occurrence of the text string in the file is highlighted. If the search is not successful, an informational message displays.
7. Click **Replace** to replace the next occurrence of the text in the **Find What** field with the text in the **Replace with** field.
8. Click **Replace All** to replace every instance of the text in the **Find What** field with the text in the **Replace with** field.

4Test Editor

The **4Test Editor**, also called test plan editor or the outline, lets you edit a file. When you are in the **4Test Editor**, there are a variety of text editing commands available through the **Outline** menu and the toolbar. You can also **AutoComplete** to automatically complete functions, members, application states, and data types in your script or include files.

If you have a data driven file open, which is a file with the extension `.g.t`, a special **4Test Editor** is displayed with additional items.

Find values of type

Lets you specify what types of values you want to search for in the file. The default is to search the test case for all types of values, text, numbers, and true/false (booleans). You can uncheck or check the boxes to indicate the values you need to look for.

Next Click to search down from the cursor's current location.

Prev click to search up from the cursor's current location.

Replace value with

Lets you specify the table and column you are using to replace the highlighted value.

- Table** Populated when you first create the `.g.t` file, or whenever you open the **Find/Replace Values** dialog box, or when you click the list box arrow. The text that displays depends on the names already saved in your data source. For example, if you are using a Microsoft Excel file as your data source, a list of worksheets and named ranges displays here. Select the table that contains the values you want to use to replace the highlighted value.
- Column** Blank when the **4Test Editor** is opened with a `.g.t` file. The text that displays here depends on the names already saved in your data source. Select the column that contains the values you want to replace the highlighted value.
- Replace** Click after you have selected a value to replace, and the table and column that contains the values you want to use as replacements.

Setting Up a Printer

Use the **Printer Setup** dialog box to set up a printer.

Options Click to display the printer's options dialog box.

Network Click to display the **Connect to Printer** dialog box where you can select another printer to use.

1. Click **File > Printer Setup**.
2. Select the printer.
 - To use the default printer, click **Default Printer**.
 - To use a different printer than the default, click **Specific Printer**.
3. Select the printer format.
 - Click **Portrait** to print the current document in portrait format (8.5 x 11).
 - Click **Landscape** to print the current document in landscape format (11 x 8.5 inches).
4. Select the size of the paper that you are using to print from the **Size** list.
5. Select the paper source that the printer should use from the **Source** list.
6. Click **Options** to display specific options for the selected printer.
7. Click **Network** to select a network printer.

Printing the Contents of the Active Window

Use the **Print** dialog box to print all or part of the contents of the active editor or results window.

1. Click **File > Print**. The **Print** dialog box appears.
2. Select **All lines** to print the entire file or **Selected lines only** to print only lines that you have highlighted.
3. Enter values for the left, right, top, and bottom margins. The default value for any margin is 0.50 inch.
4. Type literal text, variables, or a combination of both into the **Header** and **Footer** fields.
You can also enter the following into the fields:
 - `&f` to print the file name.
 - `&p` to print the page number.

For example, if you type `Script: &f` in the **Header** field of a file named `testapp.t`, the printed header will be: `Script: testapp.t`.
5. Select the output resolution from the **Print Quality** list.
6. Check the **Print Fully Expanded** check box to print the entire contents of the file. Uncheck the check box to print the file as it is shown on the screen.

7. Check the **Print Icons** check box to print icons.
8. Click **Font** to change the font family, the font size, and the style of the text.
9. Click **Setup** to display the **Printer Setup** dialog box where you can reconfigure the printer options.

Confirm Test Description Identifier Dialog Box

This dialog box is displayed when cut-and-paste editing operations lead to unresolved test description identifiers. For example, this dialog appears if you cut a test description and then pasted it back into the same place. Outwardly, nothing has changed, but the act of cutting and pasting causes a confirmation as to whether you want to retain the original test ID or not.

Glossary

This section provides an alphabetical list of terms that are related to Silk Test Classic and their descriptions.

4Test Classes

Classes are the core of object-oriented languages such as Visual Basic or 4Test. Each GUI object is an instance of a class of objects. The class defines the actions, or methods, that can be performed on all objects of a given type. For example, in 4Test the `PushButton` class defines the methods that can be performed on all pushbuttons in your application. The methods defined for pushbuttons work only on pushbuttons, not on radio lists.

The class also defines the data, or properties, of an object. In 4Test and Visual Basic, you can set or retrieve the value of a property directly using the dot operator and a syntax similar to standard Visual Basic.

4Test-Compatible Information or Methods

Information or methods that can be passed by value in 4Test prototypes.

Abstract Windowing Toolkit

The Abstract Windowing Toolkit (AWT) is a library of Java GUI object classes that is included with the Java Development Kit from Sun Microsystems. The AWT handles common interface elements for windowing environments including Windows.

The AWT contains the following set of GUI components:

- Button
- CheckBox
- CheckBox Group (RadioList)
- Choice (PopupList)
- Label (StaticText)
- List (ListBox)
- Scroll Bar
- Text Component (TextField)
- Menu

accented character

A character that has a diacritic attached to it.

agent

The agent is the software process that translates the commands in your scripts into GUI-specific commands. It is the agent that actually drives and monitors the application you are testing.

applet

A Java program designed to run inside a Java-compatible Web browser, such as Netscape Navigator.

application state

The state you expect your application to be in at the beginning of a test case. This is in addition to the conditions required for the base state.

attributes

In the test plan editor, attributes are site-specific characteristics that you can define for your test plan and assign to test descriptions and group descriptions. Each attribute has a set of values. For example, you define the Developer attribute and assign it the values of `Kate`, `Ned`, `Paul`, and `Susan`, the names of the QA engineers in your department.

Attributes are useful for grouping tests, in that you can run or report on parts of the test plan that have a given attribute value. For example, all tests that were developed by `Bob` can be executed as a group.

In Silk Test Classic, an attribute is a characteristic of an application that you verify in a test case. Attributes are used in the **Verify Window** dialog box, which is available only for projects or scripts that use the Classic Agent.

Band (.NET)

Each level in the grid hierarchy has one band object created to represent it.

base state

The known, stable state you expect the application to be in at the start of each test case.

bidirectional text

A mixture of characters that are read from left to right and characters that are read from right to left. Most Arabic and Hebrew characters, for example, are read from right to left, but numbers and quoted western terms within Arabic or Hebrew text are read from left to right.

Bytecode

The form of Java code that the Java Virtual Machine reads. Other compiled languages use compilers to translate their code into native code, also called machine code, that runs on a particular operating system. By contrast, Java compilers translate Java programs into bytecode, an intermediate form of code that is slower than compiled code, but that can theoretically run on any hardware equipped with a Java Virtual Machine.

call stack

A call stack is a listing of all the function calls that have been called to reach the current function in the script you are debugging.

In debugging mode, a list of functions and test cases which were executing at the time at which an error occurred in a script. The functions and test cases are listed in reverse order, from the last one executed back to the first.

child object

Subordinate object in the GUI hierarchy. A child object is either logically associated with, or physically contained by, its parent object. For example, the File menu, as well as all other menus, are physically contained by the main window.

class

GUI object type. The class determines which methods can operate on an object. Each object in the application is an instance of a GUI class.

class library

A collection of related classes that solve specific programming problems. The Java Abstract Windowing Toolkit (AWT) and Java Foundation Class (JFC) are examples of Java class libraries.

class mapping

Association of nonstandard custom objects with standard objects understood by Silk Test Classic.

Classic 4Test

Classic 4Test is a test scripting language that was supported with Silk Test Classic 16.0 or prior. With Silk Test Classic 16.5 or later, Silk Test Classic supports only Visual 4Test.



Note: With Silk Test Classic 16.0 or prior, you can easily convert your Classic 4Test scripts into Visual 4Test scripts by selecting **Edit > Visual 4Test** from the Silk Test Classic menu. With Silk Test Classic 16.5 or later, you can no longer convert your Classic 4Test scripts into Visual 4Test scripts. You can use an older version of Silk Test Classic to convert the scripts, or you can contact support.

client area

The internal area of a window not including scroll bars, title bar, or borders.

custom object

Nonstandard object that Silk Test Classic does not know how to interact with.

data-driven test case

A special kind of test case that receives many combinations of data from 4Test functions/test plan.

data member

Variable defined within a class or window declaration. The value of a data member can be an expression, but it is important to keep in mind that data members are resolved (assigned values) during compilation. If the expression for the data member includes variables that will change at run-time, then you must use a property instead of that data member.

declarations

See *Window Declarations*.

DefaultBaseState

Built-in application state function that returns your application to its base state. By default, the built-in `DefaultBaseState` ensures that the application is running and is not minimized, the main window of the application is open, and all other windows, for example dialog boxes and message boxes, are closed.

diacritic

1. Any mark placed over, under, or through a Latin-based character, usually to indicate a change in phonetic value from the unmarked state.
2. A character that is attached to or overlays a preceding base character.

Most diacritics are non-spacing characters that don't increase the width of the base character.

Difference Viewer

Dual-paned display-only window that lists every expected value in a test case and its corresponding actual value. Highlights all occurrences where expected and actual values differ. You display the **Difference Viewer** by selecting the box icon in the results file.

double-byte character set (DBCS)

A double-byte character set, which is a specific type of multibyte character set, includes some characters that consist of 1 byte and some characters that consist of 2 bytes.

dynamic instantiation

This special syntax is called a dynamic instantiation and is composed of the class and tag or locator of the object. For example, if there is not a declaration for the **Find** dialog box of the Text Editor application, the syntax required to identify the object looks like the following:

- Classic Agent:

```
MainWin("Text Editor|&D:\PROGRAM FILES  
  \<SilkTest install directory>\SILKTEST\TEXTEDIT.EXE").DialogBox("Find")
```

- Open Agent:

```
/MainWin[@caption='Untitled - Text Editor']//DialogBox[@caption='Find']
```

The general syntax of this kind of identifier is:

- Classic Agent:

```
class("tag").class("tag"). ...
```

- Open Agent:

```
class('locator').class('locator'). ...
```

With the Classic Agent, the recorder uses the multiple-tag settings that are stored in the **Record Window Declarations** dialog box to create the dynamic tag. In the Classic Agent example shown above, the tag for the Text Editor contains its caption as well as its window ID. For additional information, see *About Tags*.

dynamic link library (DLL)

A library of reusable functions that allow code, data, and resources to be shared among programs using the module. Programs are linked to the module dynamically at runtime.

enabling

Altering program code to handle input, display, and editing of bidirectional or double-byte languages, such as Arabic and Japanese.

exception

Signal that something did not work as expected in a script. Logs the error in the results file.

frame file

See test frame file.

fully qualified object name

Name that uniquely identifies a GUI object. The actual format depends on whether or not a window declaration has been previously recorded for the object and its ancestors.

group description

In the test plan editor, one or more lines in an outline that describe a group of tests, not a single test. Group descriptions by default are displayed in black.

handles

A handle is an identification code provided for certain types of object so that you can pass it to a function that needs to know which object to manipulate.

hierarchy of GUI objects

Parent-child relationships between GUI objects.

host machine

A host machine is a system that runs the Silk Test Classic software process in which you develop, edit, compile, run, and debug 4Test scripts and test plans.

Host machines are always Windows systems.

hotkey

The following table lists the available hotkeys and accelerator keys for each menu:

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
Breakpoint	Toggle	Alt+B+T	F5
	Add	Alt+B+A	-
	Delete	Alt+B+D	-
	Delete All	Alt+B+E	-
Debug	Abort	Alt+D+A	-
	Exit	Alt+D+X	-
	Finish Function	Alt+D+F	-
	Reset	Alt+D+E	-
	Run and Debug/Continue	Alt+D+R	F9
	Run to Cursor	Alt+D+C	Shift+F9
	Step Into	Alt+D+I	F7
	Step Over	Alt+D+S	F8
	Edit	Undo	Alt+E+U
Redo		Alt+E+R	Ctrl+Y
Cut		Alt+E+T	Ctrl+X
Copy		Alt+E+C	Ctrl+C
Paste		Alt+E+P	Ctrl+V
Delete		Alt+E+D	Del
Find		Alt+E+F	Ctrl+F

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
	Find Next	Alt+E+N	F3
	Replace	Alt+E+E	Ctrl+R
	GoTo Line	Alt+E+G	Ctrl+G
	GoTo Definition	Alt+E+O	F12
	Find Error	Alt+E+I	F4
	Data Driven	-	-
File	New	Alt+F+N	Ctrl+N
	Open	Alt+F+O	Ctrl+O
	Save	Alt+F+S	Ctrl+S
	Save As	Alt+F+A	-
	Save All	Alt+F+L	-
	New Project	Alt+F+W	-
	Open Project	Alt+F+E	-
	Close Project	Alt+F+J	-
	Run	Alt+F+R	-
	Debug	Alt+F+D	-
	Check out	Alt+F+T	Ctrl+T
	Check in	Alt+F+K	Ctrl+K
	Print	Alt+F+P	Ctrl+P
	Printer Setup	Alt+F+I	-
	# operation file-name	Alt+F+#	Alt+F+#
	Exit	Alt+F+X	Alt+F4
Help	Help Topics	Alt+H+H	-
	Library Browser	Alt+H+L	-
	Tutorials	Alt+H+T	-
	About Silk Test Classic	Alt+H+A	-
Include	Open	Alt+I+O	-
	Open All	Alt+I+P	-
	Close	Alt+I+C	-
	Close All	Alt+I+L	-
	Save	Alt+I+S	-
	Acquire Lock	Alt+I+A	-
	Release Lock	Alt+I+R	-
Options	General	Alt+O+G	-
	Editor Font	Alt+O+D	-
	Editor Colors	Alt+O+E	-

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
	Runtime	Alt+O+T	-
	Agent	Alt+O+A	-
	Extensions	Alt+O+X	-
	Recorder	Alt+O+R	-
	Class Map	Alt+O+M	-
	Property Sets	Alt+O+P	-
	Silk Central URLs	Alt+O+U	-
	Open Options Set	Alt+O+O	-
	Save Options Set	Alt+O+S	-
	Close Options Set	Alt+O+C	-
	# option-file-name	Alt+O+#	-
Outline	Move Left	Alt+L+V	Alt+Left Arrow
	Move Right	Alt+L+R	Alt+Right Arrow
	Transpose Up	Alt+L+A	Alt+Up Arrow
	Transpose Down	Alt+L+S	Alt+Down Arrow
	Expand	Alt+L+E	Ctrl++
	Expand All	Alt+L+X	Ctrl+*
	Collapse	Alt+L+O	Ctrl+-
	Collapse All	Alt+L+L	Ctrl+/-
	Comment	Alt+L+M	Alt+M
	Uncomment	Alt+L+N	Alt+N
Project	View Explorer	Alt+P+V	-
	Align	Alt+P+A	-
	&Left	Alt+P+L	-
	&Right	Alt+P+R	-
	Project Description	Alt+P+O	-
	Add File	Alt+P+D	-
	Remove File	Alt+P+R	-
Record	Window Declarations	Alt+R+W	Ctrl+W
	Application State	Alt+R+S	-
	Testcase	Alt+R+	Ctrl+E
	Method	Alt+R+T	-
	Actions	Alt+R+A	-
	Class	Alt+R+C	-
	Window Identifiers	Alt+R+I	Ctrl+I
	Window Locations	Alt+R+L	-

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
Results	Select	Alt+T+S	-
	Merge	Alt+T+M	-
	Delete	Alt+T+D	-
	Extract	Alt+T+E	-
	Export	Alt+T+X	-
	Convert to Plan	Alt+T+C	-
	Compact	-	-
	Show Summary	Alt+T+H	-
	Hide Summary	Alt+T+I	-
	View Options	Alt+T+V	-
	Goto Source	Alt+T+G	-
	View Differences	Alt+T+W	-
	Update Expected Value	Alt+T+U	-
	Pass/Fail Report	Alt+T+P	-
	Mark Failures in Plan	Alt+T+F	-
	Compare Two Results	Alt+T+O	-
	Next Result Difference	Alt+T+N	-
Next Error Difference	Alt+T+r	-	
Run	Compile	Alt+U+C	Alt+F9
	Compile all	-	-
	Run All Tests	Alt+U+R	F9
	Debug	Alt+U+D	Ctrl+F9
	Application State	Alt+U+A	Alt+A
	Testcase	Alt+U+T	Alt+T
	Show Status	Alt+U+S	-
Abort	Alt+U+B	LShift+RShift	
Testplan	Goto Script	Alt+T+G	-
	Detail	Alt+T+D	-
	Insert Template	Alt+T+I	-
	Completion Report	Alt+T+C	-
	Mark	Alt+T+M	-
	Mark All	Alt+T+A	-
	Unmark	Alt+T+U	-
	Unmark All	Alt+T+N	-
	Mark by Query	Alt+T+Q	-
	Mary by Named Query	Alt+T+R	-

Menu Name	Command with Hotkey	Hotkey	Accelerator Key
Tools	Find Next Mark	Alt+T+F	-
	Define Attributes	Alt+T+E	-
	Manual tests	Alt+T+T	-
	Link Tester	Alt+S+L	-
	Start Silk Performer	Alt+S+P	-
	Data Drive Testcase	Alt+S+D	-
	Enable Extensions	Alt+S+E	-
View/Transcript	Silk Central Test Manager	Alt+S+H	-
	Expression	Alt+V+E	-
	Global Variables	Alt+V+G	-
	Local Variables	Alt+V+L	-
	Expand Data	Alt+V+X	-
	Collapse Data	Alt+V+C	-
	Module	Alt+V+M	-
	Breakpoints	Alt+V+B	-
	Call Stack	Alt+V+L	-
	Transcript	Alt+V+T	-
Window	Tile Vertically	Alt+W+T	-
	Tile Horizontally	Alt+W+H	-
	Cascade	Alt+W+C	-
	Arrange Icons	Alt+W+E	-
	Close All	Alt+W+L	-
	Next	Alt+W+N	F6
	Previous	Alt+W+P	Shift+F6
	# file-file-name	Alt+W+#	-
Workflows	Basic	Alt+K+B	-
	Data Driven	Alt+K+D	-

Hungarian notation

Naming convention in which a variable's name begins with one or more lowercase letters indicating its data type. For example, the variable name *sCommandLine* indicates that the data type of the variable is *STRING*.

identifier

Name used in test scripts to refer to an object in the application. Logical, GUI-independent name. Identifier is mapped to the tag in a window declaration.

include file

File that contains window declarations and can contain constant, variable, and other declarations.

internationalization or globalization

The process of developing a program core whose feature design and code design don't make assumptions based on a single language or locale and whose source code base simplifies the creation of different language editions of a program.

Java Database Connectivity (JDBC)

Java API that enables Java programs to execute SQL statements and interact with any SQL-compliant database. Often abbreviated as JDBC.

Java Development Kit (JDK)

A free tool for building Java applets and full-scale applications. This is an environment which contains development and debugging tools, and documentation. Often abbreviated as JDK.

Java Foundation Classes (JFC)

Sun Microsystem's and Netscape's class library designed for building visual applications in Java. Often abbreviated as JFC.

JFC consists of a set of GUI components named Swing that adopt the native look and feel of the platforms they run on.

Java Runtime Environment (JRE)

Sun Microsystem's execution-only subset of its Java Development Kit. The Java Runtime Environment (JRE) consists of the Java Virtual Machine, Java Core Classes, and supporting files, but contains no compiler, no debugger, and no tools.

The JRE provides two virtual machines: `JRE.EXE` and `JREW.EXE`. The only difference is that `JREW` does not have a console window.

Java Virtual Machine (JVM)

Software that interprets Java code for a computer's operating system. A single Java applet or application can run unmodified on any operating system that has a virtual machine, or VM.

JavaBeans

Reusable software components written in Java that perform a single function. JavaBeans can be mixed and matched to build complex applications because they can identify each other and exchange information.

JavaBeans are similar to ActiveX controls and can communicate with ActiveX controls. Unlike ActiveX, JavaBeans are platform-independent.

Latin script

The set of 26 characters (A through Z) inherited from the Roman Empire that, together with later character additions, is used to write languages throughout Africa, the Americas, parts of Asia, Europe, and Oceania. The Windows 3.1 Latin 1 character set covers Western European languages and languages that use the same alphabet. The Latin 2 character set covers Central and Eastern European languages.

layout

The order and spacing of displayed text.

levels of localization

The amount of translation and customization necessary to create different language editions. The levels, which are determined by balancing risk and return, range from translating nothing to shipping a completely translated product with customized features.

load testing

Testing that determines the actual, which means not simulated, impact of multi-machine operations on an application, the server, the network, and all related elements.

localization

The process of adapting a program for a specific international market, which includes translating the user interface, resizing dialog boxes, customizing features if necessary, and testing results to ensure that the program still works.

localize an application

To make an application suitable for a specific locale: for example, to include foreign language strings for an international site.

locator

This functionality is supported only if you are using the Open Agent.

The locator is the actual name of an object, to which Silk Test Classic maps the identifier for a GUI object. You can use locator keywords to create scripts that use dynamic object recognition and window declarations.

logical hierarchy

The hierarchy that is implied from the visible organization of windows as they display to the user.

manual test

In the testplan editor, a manual test is a test that is documented but cannot be automated and, therefore, cannot be run within the test plan. You might choose to include manual tests in your test plan in order to centralize the testing process. To indicate that a test description is implemented manually, you use the keyword value `manual` in the testcase statement.

mark

In the testplan editor, a mark is a technique used to work with one or more tests as a group. A mark is denoted by a black stripe in the margin bar of the test plan. Marks are temporary and last only as long as the current work session. Tests that are marked can be run or reported on independently as a subset of the total plan.

master plan

In the testplan editor, that portion of a test plan that contains only the top few levels of group descriptions. You can expand, which means display, the sub-plans of the master plan, which contain the remaining levels of group description and test description. The master plan/sub-plan approach allows multi-user access to a test plan, while at the same time maintaining a single point of control for the entire project. A master plan file has a `.pln` extension.

message box

Dialog box that has only static text and pushbuttons. Typically, message boxes are used to prompt a user to verify an action, such as `Save changes before closing?`, or to alert a user to an error.

method

Operation, or action, to perform on a GUI object. Each class defines its own set of methods. Methods are also inherited from the class's ancestors.

minus (-) sign

In a file, an icon that indicates that all information is displayed. Click on the minus sign to hide the information. The minus sign becomes a plus sign.

modal

A dialog box that presents a task that must be completed before continuing with the application. No other part of the application can be accessed until the dialog box is closed. Often used for error messages.

modeless

A dialog box that presents a simple or ongoing task. May be left open while accessing other features of the application, for example, a search dialog box.

Multibyte Character Set (MBCS)

A mixed-width character set, in which some characters consist of more than 1 byte.

Multiple Application Domains (.NET)

The .NET Framework supports multiple application domains. A new application domain loads its own copies of the common language runtime DLLs, data structure, and memory pools. Multiple application domains can exist in one operation system process.

negative testing

Tests that deliberately introduce an error to check an application's behavior and robustness. For example, erroneous data may be entered, or attempts made to force the application to perform an operation that it should not be able to complete. Generally a message box is generated to inform the user of the problem.

nested declarations

Indented declarations that denote the hierarchical relationships of GUI objects in an application.

No-Touch (.NET)

No-Touch deployment allows Windows Forms applications, which are applications built using Windows Forms classes of the .NET Framework, to be downloaded, installed, and run directly on the machines of the user, without any alteration of the registry or shared system components.

object

The principal building block of object-oriented programs. Each object is a programming unit consisting of data and functionality. Objects inherit their methods and properties from the classes to which they belong.

outline

In the test plan editor, a structured, usually hierarchical model that describes the requirements of a test plan and contains the statements that implement the requirements. The outline supports automatic, context-sensitive coloring of test plan elements.

In Silk Test Classic, the outline is a 4Test editor mode that supports automatic, context-sensitive coloring and indenting of 4Test elements.

Overloaded method

A method that you call with different sets of parameter lists. Overloaded methods cause naming conflicts which must be resolved to avoid runtime errors when testing Java applications.

Example of an overloaded method	How Java support resolves the naming conflict
setBounds(int i1, int i2, int i3, int i4)	setBounds(int i1, int i2, int i3, int i4)
setBounds(RECT r1)	setBounds_2(RECT r1)

parent object

Superior object in the GUI hierarchy. A parent object is either logically associated with or physically contains a subordinate object, the child. For example, the main window physically contains the File menu as well as all other menus.

performance testing

Testing to verify that an operation in an application performs within a specified, acceptable period of time. Alternately, testing to verify that space consumption of an application stays within specified limits.

physical hierarchy (.NET)

The window handle hierarchy as implemented by the application developer.

plus (+) sign

In a file, an icon that indicates that there is hidden information. You can show the information by clicking on the plus sign. The plus sign becomes a minus sign.

polymorphism

Different classes or objects performing the same named task, but with different execution logic.

project

Silk Test Classic projects organize all the resources associated with a test set and present them visually in the **Project Explorer**, making it easy to see, manage, and work within your test environment.

Silk Test Classic projects store relevant information about your project, including references to all the resources associated with a test set, such as plans, scripts, data, option sets, .ini files, results, and frame/include files, as well as configuration information, Editor settings, and data files for attributes and queries. All of this information is stored at the project level, meaning that once you add the appropriate files to your project and configure it once, you may never need to do it again. Switching among projects is easy - since you need to configure the project only once, you can simply open the project and run your tests.

properties

Characteristics, values, or information associated with an object, such as its state or current value.

query

User-selected set of characteristics that are compared to the attributes, symbols, or execution characteristics in a test plan. When the set of characteristics matches a test, the test is marked. This is called marking by query. For example, you might run a query in order to mark all tests that are defined in the `find.t` script and that were created by the developer named Bob.

recovery system

A built-in, automatic mechanism to ensure the application is in a known state. If the application is not in the expected state, a message is logged to the results file and the problem is corrected. The recovery system is invoked before and after each test case is executed.

regression testing

A set of baseline tests that are run against each new build of an application to determine if the current build has regressed in quality from the previous one.

results file

A file that lists information about the scripts and test cases that you ran. In the testplan editor, a results file also lists information about the test plan that you ran; the format of a results file mimics the outline format of the test plan it derives from. The name of the results file is `script-name.res` or `testplan-name.res`.

script

A collection of related 4Test test cases and functions that reside in a script file.

script file

A file that contains one or more related test cases. A script file has a `.t` extension, such as `find.t`.

side-by-side (.NET)

Side-by-side execution is the ability to install multiple versions of code so that an application can choose which version of the common language runtime or of a component it uses.

Simplified Chinese

The Chinese alphabet that consists of several thousand ideographic characters that are simplified versions of traditional Chinese characters.

Single-Byte Character Set (SBCS)

A character encoding in which each character is represented by 1 byte. Single byte character sets are mathematically limited to 256 characters.

smoke test

Tests that constitute a quick set of acceptance tests. They are often used to verify a minimum level of functionality before either accepting a new build into source control or continuing QA with more in-depth, time-consuming testing.

Standard Widget Toolkit (SWT)

The Standard Widget Toolkit (SWT) is a graphical widget toolkit for the Java platform. SWT is an alternative to the AWT and Swing Java GUI toolkits provided by Sun Microsystems. SWT was originally developed by IBM and is maintained by the Eclipse Foundation in tandem with the Eclipse IDE.

statement

In the testplan editor, lines that implement the requirements of a test plan. The testplan editor has the following statements:

- testcase
- script
- testdata
- include
- attribute

Statements consist of one of the preceding keywords followed by a colon and a value.

In Silk Test Classic, a statement is a method or function call or 4Test flow-of control command, such as `if . . then`, that is used within a 4Test test case.

status line

Area at the bottom of the window that displays the status of the current script, the line and column of the active window (if any), and the name of the script that is currently running. When the cursor is positioned over the toolbar, it displays a brief description of the item.

stress testing

Tests that exercise an application by repeating the same commands or operation a large number of times.

subplan

Test plan that is referenced by another test plan, normally the master test plan, by using an include statement. Portion of a test plan that resides in a separate file but can be expanded inline within its master plan. A subplan may contain the levels of group description and test description not covered in the master plan. A subplan can inherit information from its master plan. You add a subplan by inserting an include statement in the master plan. A subplan file has a `.pln` extension, as in `subplan-name.pln`.

suite

A file that names any number of 4Test test script files. Instead of running each script individually, you run the suite, which executes in turn each of your scripts and all the test cases it contains.

Swing

A set of GUI components implemented in Java that are based on the Lightweight UI Framework. Swing components include:

- Java versions of the existing Abstract Windowing Toolkit (AWT) components, such as Button, Scrollbar, and List.
- A set of high-level Java components, such as tree-view, list-box, and tabbed-pane components.

The Swing tool set lets you create a set of GUI components that automatically implements the appearance and behavior of components designed for any OS platform, but without requiring window-system-specific code.

Swing components are part of the Java Foundation Class library beginning with version 1.1.

symbols

In the testplan editor, used in a test plan to pass data to 4Test test cases. A symbol can be defined at a level in the test plan where it can be shared by a group of tests. Its values are actually assigned at either the group or test description level, depending on whether the values are shared by many tests or are unique to a single test. Similar to a 4Test identifier, except that its name begins with a `$` character.

tag

This functionality is available only for projects or scripts that use the Classic Agent.

The actual name or index of the object as it is displayed in the GUI. The name by which Silk Test Classic locates and identifies objects in the application.

target machine

A target machine is a system (or systems) that runs the 4Test Agent, which is the software process that translates the commands in your scripts into GUI-specific commands, in essence, driving and monitoring your applications under test.

One Agent process can run locally on the host machine, but in a networked environment, the host machine can connect to any number of remote Agents simultaneously or sequentially.

Target machines can be Windows systems.

template

A hierarchical outline in the testplan editor that you can use as a guide when creating a new test plan. Based on the window declarations in the frame file.

test description

In the testplan editor, a terminal point in an outline that specifies a test case to be executed. Test descriptions by default are displayed in blue.

test frame file

Contains all the data structures that support your scripts:

- window declarations
- user-defined classes
- utility functions
- constants
- variables
- other include files

test case

In a script file, an automated test that ideally addresses one test requirement. Specifically, a 4Test function that begins with the `testcase` keyword and contains a sequence of 4Test statements. It drives an application to the state to be tested, verifies that the application works as expected, and returns the application to its base state.

In a test plan, a testcase is a keyword whose value is the name of a test case defined in a script file. Used in an assignment statement to link a test description in a test plan with a 4Test test case defined in a script file.

Test case names can have a maximum of 127 characters. When you create a data driven test case, Silk Test Classic truncates any test case name that is greater than 124 characters.

test plan

In general, a document that describes test requirements. In the testplan editor, a test plan is displayed in an easy-to-read outline format, which lists the test requirements in high-level prose descriptions. The structure can be flat or many levels deep. Indentation indicates the level of detail. A test plan also contains statements, which are keywords and values that implement the test descriptions by linking them to 4Test test cases. Large test plans can be divided into a master plan and one or more sub plans. A test plan file has a `.pln` extension, such as `find.pln`.

TotalMemory parameter

Total amount of memory available to the Java interpreter. This is the value returned from the `java.lang.Runtime.totalMemory()` method.

Traditional Chinese

The set of Chinese characters, used in such countries or regions as Hong Kong SAR, China Singapore, and Taiwan, that is consistent with the original form of Chinese ideographs that are several thousand years old.

variable

A named location in which you can store a piece of information. Analogous to a labeled drawer in a file cabinet.

verification statement

4Test code that checks that an application is working by comparing an actual result against an expected (baseline) result.

Visual 4Test

Visual 4Test is the test scripting language used by Silk Test Classic.

window declarations

Descriptions of all the objects in the application's graphical user interface, such as menus and dialog boxes. Declarations are stored in an include file which has a .inc extension, typically the `frame.inc` file.

window part

Predefined identifiers for referring to parts of the window. Associated with common parts of `MoveableWin` and `Control` classes, such as `LeftEdge`, `MenuBar`, `ScrollBar`.

XPath

The XML Path Language (XPath) models an XML document as a tree of nodes and enables you to address parts of the XML document. XPath uses a path notation to navigate through the hierarchical structure of the XML document. Dynamic object recognition uses a `Find` or `FindAll` function and an XPath query to locate the objects that you want to test.

Index

- .NET, Open Agent
 - testing applications 267
- # operator
 - testplan editor 122
- + and - operators
 - rules 133

- 4Test
 - versus native Java controls 290
- 4Test classes
 - definition 571
- 4Test code
 - marking as GUI specific 411
- 4Test Editor
 - about 568
 - compatible information or methods 571
 - not enough characters displayed 549
- 4Test methods
 - comparing with native methods 297
- 4test.inc
 - relationship with messages sent to the result file 549

A

- Abstract Windowing Toolkit
 - overview 571
- accented characters
 - definition 571
- Accessibility
 - enabling for the Open Agent 389
 - improving object recognition 389
 - Open Agent 389
- accessing
 - files in projects 67
- accessing data
 - member-of operator 387
- acquiring locks
 - test plans 120
- active object
 - highlight during recording 169
- active script
 - running 445
- active suite
 - running 445
- Add Breakpoint
 - dialog box 482
- add-ons
 - Google Chrome 346
 - Mozilla Firefox 349
- adding
 - keywords 370
- adding comments
 - test plan editor 122
- adding files
 - projects 73
- adding folders

- projects 74
- adding information to the beginning of a file
 - using file functions 550
- adding keywords
 - keyword-driven tests 369
- adding method to TextField class
 - example 403
- adding properties
 - recorder 541
- adding Tab method to DialogBox class
 - example 404
- admin rights
 - installing 546
 - running 546
- Adobe Flex
 - adding configuration information 243
 - Adobe Air support 238
 - automation support for custom controls 260
 - automationName property 245
 - coding containers 248
 - containers 248
 - creating applications 244
 - defining custom controls 255
 - loading at run-time 242
 - multiview containers 248
 - passing parameters 243
 - passing parameters at runtime 243
 - passing parameters before runtime 243
 - run-time loading 242
 - security settings 234
 - select method, overview 238
 - select method, setting 247
 - testing playback 250
- advanced options
 - Agent Options dialog box 500
- advanced techniques
 - Open Agent 378
- agent
 - definition 571
- agent not responding
 - error message 520
- agent options
 - differences 56
 - Open Agent 26
 - setting for Web testing 93
 - setting window timeout 48
- Agent Options
 - dialog box 494
- Agent Options dialog box
 - advanced options 500
 - bitmap agent options 499
 - bitmap options 498
 - close options 497
 - compatibility options 502
 - other options 501
 - replay options 446, 508
 - synchronization options 499
 - timing options 495

- verification options 496
- agent support
 - Java AWT 282
 - Swing 282
- AgentClass class
 - classes for non-window objects 394
- agents
 - assigning to window declarations 26
 - comparison, classes 59
 - configuring ports, information service 49
 - connecting to default 46
 - differences, classes 59
 - differences, object recognition 57
 - differences, options 56
 - driving associated applications simultaneously 209
 - options 26
 - parameter comparison 63
 - parameters 63
 - record functionality 47
 - setting default 45
 - setting default, Runtime Options 45
 - setting default, toolbar 46
 - supported methods 64
 - supported SYS functions 64
 - unable to connect 521
 - using both agents 46
- AJAX applications
 - script hangs 355
- Android
 - configuring emulator 303
 - enabling USB-debugging 303
 - hybrid applications 302
 - installed apps, testing 311
 - installing USB drivers 302
 - mobile native applications, prerequisites 301
 - mobile web applications, prerequisites 301
 - recommended settings 303
 - recording test cases 53, 146
 - releasing devices 316
 - releasing devices, recording 316
 - releasing devices, replay 316
 - testing 301
 - troubleshooting 317
- Android emulator
 - configuring 303
- animation mode
 - test cases 447
- AnyWin class
 - cannot extend class 542
- Apache Flex
 - automationIndex property 245
 - automationName property 245
 - class definition file 262
 - Component Explorer 250
 - Component Explorer, launching 250
 - controls are not recognized 519
 - custom controls 416
 - customizing scripts 235, 254
 - enabling your application 240
 - exception values 238
 - Flash player settings 234
 - FlexDataGrid control, selecting items 239
 - initializing, applications 249
 - linking automation packages 240
 - locator attributes 236
 - overview 233
 - precompiling the application 241
 - prerequisites 233, 250
 - recording, applications 249
 - sample test case 252
 - select method, setting 247
 - styles 235
 - testing 232
 - testing custom controls 254
 - testing custom controls using automation support 259
 - testing custom controls using dynamic invoke 258
 - testing multiple applications, same web page 237
 - troubleshooting 519
 - using dynamic invoke 236
 - verifying scripts 235, 254
 - workflow 249
- Apache Flex applications
 - custom attributes 166
- API playback
 - comparing to, native playback 339
- appearance
 - verifying by using a bitmap 180
- appending test cases
 - data-driven scripts 198
- Apple Safari
 - connection string 341
 - information service, installing 309, 310, 342, 344
 - limitations 343
 - prerequisites 342
 - running multiple tests 344
 - support 326
 - testing 342
- applet
 - definition 572
- applets
 - controls not recognized 291, 534
- application behavior differences
 - supporting 407
- application configurations
 - keyword-driven tests 367
- application state
 - definition 572
- application states
 - behavior of based on NONE 177
 - overview 176
 - recording 150
 - testing 154
- applications
 - configuring 52
 - preparing for automated testing 553
 - single and remote 204
- applications with invalid data
 - testing 188
- applying masks
 - exclude all differences 468
 - excluding differences 467
 - excluding selected areas 467
- AppStateList
 - using 433

- array indexing
 - indexed values in test scripts 290
 - assigning agents
 - window declarations 26
 - assigning attributes
 - Testplan Detail dialog box 134
 - attaching
 - comments, result sets 470
 - attribute definitions
 - modifying 134
 - attribute types
 - dynamic object recognition 162
 - editing 162
 - Oracle Forms 286
 - attribute values
 - defining 133
 - editing 162
 - attributes
 - assigning to test plans 133
 - custom 165, 283, 298
 - Define Attributes dialog box 141
 - defining 133
 - defining for existing classes 399
 - definition 399, 572
 - modifying definition 134
 - New Attribute dialog box 142
 - syntax 399
 - test plans 131
 - verification 399
 - verifying 398
 - attributes and values
 - overview 131
 - autocomplete
 - using 430
 - AutoComplete
 - AppStateList 433
 - customizing MemberList 431
 - DataTypeList 434
 - FAQs 432
 - FunctionTip 434
 - MemberList 434
 - overview 430
 - turning off 433
 - automated testing
 - making locators easier to recognize 553
 - automatically generated code
 - data-driven test cases 185
 - AWT
 - overview 571
 - predefined classes 292
 - recording menus 296
 - AWT classes
 - predefined 292
- B**
- band (.NET)
 - definition 572
 - base state
 - about 99
 - definition 572
 - keyword-driven tests 367
 - modifying 100
 - based on NONE
 - application state behavior 177
 - baseline bitmaps
 - result bitmaps, graphical differences 469
 - basic workflow
 - Open Agent 51
 - troubleshooting issues, Open Agent 519
 - basic workflow issues
 - troubleshooting, Open Agent 519
 - Beans
 - definition 581
 - best practices
 - scripts, creating 153
 - bi-directional languages
 - support 426
 - BiDi text
 - definition 572
 - bidirectional text
 - definition 572
 - bitmap comparison
 - excluding parts of the bitmap 466
 - rules 464
 - bitmap differences
 - scanning 469
 - bitmap options
 - agent options 499
 - Agent Options dialog box 498
 - Bitmap Tool
 - applying a mask 467
 - baseline and result bitmaps 464
 - capturing a bitmap 461
 - capturing bitmaps 461
 - comparing bitmaps 463
 - designate bitmap as baseline 465
 - designating a bitmap as a results file 465
 - editing masks 467
 - exiting from scan mode 465
 - mask prerequisites 467
 - moving to the next or previous difference 470
 - opening bitmap files 466
 - overview 460
 - saving captured bitmaps 463
 - starting 466
 - starting from icon 466
 - starting from the results file 466
 - starting from the Run dialog box 466
 - un-setting a designated bitmap 465
 - using masks 466
 - zooming windows 465
 - bitmaps
 - agent options 499
 - analyzing 460
 - analyzing for differences 469
 - baseline 464
 - baseline vs result, graphical differences 469
 - Bitmap Tool overview 460
 - capturing during recording 462
 - capturing Zoom window in scan mode 462
 - comparing baseline and result bitmaps 465
 - comparison command rules 464
 - designate as baseline 465

- designate as results file 465
- exiting from scan mode 465
- functions 464
- options 498
- result 464
- saving captured bitmaps 463
- saving masks 469
- scanning differences 469
- showing areas of difference 469
- starting the Bitmap Tool 466
- statistics 465
- un-setting designated bitmaps 465
- verifying 180
- verifying appearance 180
- viewing statistics 465
- when to use the Bitmap Tool 461, 555
- zooming in on differences 470

- both agents
 - creating script 46

- break statements
 - incorrect usage 486

- Breakpoint dialog box 482

- breakpoints
 - about 481
 - Add Breakpoint dialog box 482
 - Delete Breakpoint dialog box 482
 - deleting 482
 - setting 481
 - setting temporary 481
 - viewing 481

- browser configuration settings
 - xBrowser 340

- browser extensions
 - disabling 91

- browser recognized
 - as client/server application 267

- browser specifiers
 - web applications 329

- browser test failure
 - troubleshooting 555

- browser type
 - setting 330
 - viewing current 355
 - viewing current, GetProperty 355

- browsers
 - configuring 92

- browsertype
 - using 355

- building queries
 - tables 195

- bytecode
 - definition 572

C

- call stack
 - definition 573

- Call Stack dialog box 485

- calling DLLs
 - within 4Test scripts 390

- calling nested methods
 - InvokeJava method 295

- calling Windows DLLs from 4Test
 - overview 390

- cannot double-click
 - file to open Silk Test Classic 542

- cannot extend
 - classes 542

- cannot find items
 - Classic 4Test 82, 553

- cannot open Silk Test Classic
 - by double-clicking a file 542

- cannot save files
 - projects 81, 552

- cannot start
 - Silk Test Classic 82, 552

- captions
 - GUI-specific 413

- capturing a bitmap
 - Bitmap Tool 461

- capturing bitmaps
 - during recording 462
 - Bitmap Tool 461

- categorizing test plans
 - overview 125

- changes not applied
 - include files or scripts 547

- changing element colors
 - result files 472

- Chart Settings dialog box 143

- charts
 - presenting results 475

- checking precedence
 - operators 485

- child object
 - definition 573

- Chrome
 - changing browser type for replay 330
 - configuration settings 340
 - cross-browser scripts 354
 - extensions, testing 346
 - locators 355
 - prerequisites 346
 - testing 345
 - user data directories, testing 347

- Chrome for Android
 - support 326

- class
 - definition 573

- class definition file
 - Java 291

- class hierarchy
 - 4Test (Open Agent) 397

- class library
 - definition 573

- class mapping
 - definition 573

- class methods
 - viewing in Library Browser 437

- class properties
 - NumChildren alternative 400

- classes
 - 4Test 394
 - agent comparison 59
 - declarations 410
 - defining attributes 399

- defining properties 397
- defining with Open Agent 395
- hierarchy (Open Agent) 397
- logical 397
- overview 393
- WPF 277
- xBrowser 360
- classes for non-window objects
 - AgentClass 394
 - ClipboardClass 394
 - CursorClass 394
- Classic 4Test
 - cannot find items 82, 553
 - definition 573
- Classic Agent
 - comparison to Open Agent, classes 59
 - migrating to the Open Agent 56
 - recording options, setting 507
- Classic Agent parameters
 - comparison to Open Agent 63
- CLASSPATH
 - disabling when Java is installed 297
- Click
 - mobile web 322
- client area
 - definition 573
- client/server applications
 - overview 263
- client/server testing
 - challenges 263
 - code for template.t 229
 - concurrency testing 265
 - configuration testing 266
 - configurations 205
 - functional testing 266
 - multi_cs.t script 217
 - multi-application testing 223
 - multi-testcase code template 217
 - parallel template 216
 - parallel.t script 216
 - serially 222
 - template.t explained 229
 - testing databases 223
 - types of testing 265
 - verifying tables 263
- clients
 - testing concurrently 220
- Clipboard methods
 - 4Test 420
 - code sample 420
- ClipboardClass class
 - classes for non-window objects 394
- close options
 - Agent Options dialog box 497
- closing windows
 - recovery system 104
 - specifying buttons 109
 - specifying keys 109
 - specifying menus 109
- colors
 - setting 490
- columns
 - testing in Web applications 332
- Combine Property Sets
 - dialog box 511
- Combine Testplan Queries dialog box 138
- combining
 - keywords 370
- command line
 - starting from 378
 - starting from, Silk Test Classic 378
- comments
 - attaching, result sets 470
- Compare Two Results dialog box 456
- comparing
 - result files 470
- comparing bitmaps
 - Bitmap Tool 463
- compatibility options
 - Agent Options dialog box 502
- compile errors
 - Unicode content 430
- Compiler Constants
 - dialog box 494
- compiling
 - conditional compilation 410
- compiling code
 - conditionally 410
- completion reports
 - generating for test plans 120
- Component Explorer
 - Apache Flex, launching 250
 - recording sample test case 252
 - testing 250
- concurrency
 - processing 209
- concurrency testing
 - code example 225
 - explanation of code example 226
 - overview 265
- concurrent programming
 - threads 211
- concurrently testing
 - clients 220
- conditional compilation
 - result 412
- conditionally compiling code
 - outcome 412
- configuration test failures
 - troubleshooting 267
- configuration testing
 - client/server testing 266
 - overview 266
- configuring
 - network of computers 204
- configuring applications
 - custom 362
 - Java 287
 - mobile Web 251
 - overview 52
 - standard 362
 - Web 251
- configuring ports
 - information service, clients 49

- configuring sample Web application
 - insurance company 358
- Confirm Test Description Identifier dialog box 570
- connecting
 - default agent 46
 - Open Agent, remote 203
- connection string
 - desktop browsers, remote 341
 - mobile devices 312
- contact information 21, 22
- containers
 - invisible 421
- continuous integration
 - uploading keyword libraries 376
- Control class
 - cannot extend class 542
- control is not responding 520
- controls
 - recognized as custom controls 414
 - testing for Web applications 332
 - verifying that no longer displayed 183
- Convert Results to Plan
 - dialog box 114
- converting
 - test sets to projects 71
- Create Session dialog box 139
- create test case
 - basic workflow for the Open Agent 51
- creating
 - file 567
 - keyword-driven tests 366
 - projects 51, 69, 251
 - suites 443
- creating a new project
 - insurance company Web application 357
- creating data-driven scripts
 - Specify Data Driven Script dialog box 198
- creating data-driven test cases
 - workflow 184
- creating masks
 - exclude all differences 468
 - excluding differences 467
 - excluding selected areas 467
- creating new queries
 - combining queries 135
- creating script
 - both agents 46
- creating stable locators
 - overview 164
- creating test cases
 - Open Agent 145
- cross browser testing
 - Apple Safari 342
 - Apple Safari, limitations 343
 - current browser type, viewing 355
 - FAQs 353
 - Google Chrome 345
 - Microsoft Edge 351
 - Microsoft Edge, limitations 351
 - Mozilla Firefox 348
 - object recognition 334
 - overview 326
 - recording locators 355
 - test objects 328
 - wrong timestamps, logs 355
- cross-browser testing
 - Apple Safari 342
 - Apple Safari, limitations 343
 - connection string 341
 - current browser type, viewing 355
 - FAQs 353
 - Google Chrome 345
 - Microsoft Edge 351
 - Microsoft Edge, limitations 351
 - Mozilla Firefox 348
 - object recognition 334
 - overview 326
 - recording locators 355
 - remote locations, adding 155
 - test objects 328
 - wrong timestamps, logs 355
- cross-platform methods
 - using in scripts 409
- cs.inc
 - overview 231
- current browser type
 - viewing 355
- CursorClass class
 - classes for non-window objects 394
- custom applications
 - configuring 362
- custom attributes
 - about 165, 283, 298
 - Apache Flex applications 166
 - setting to use in locators 508
 - Web applications 167
 - Windows Forms applications 168
 - WPF applications 168
- custom classes
 - filtering 421
- custom controls
 - creating custom classes 418
 - dialog box 419, 509
 - dynamic invoke, FAQs 416
 - invoke call returns unexpected string 416
 - Java 281
 - managing 417
 - overview 415
 - supporting 414, 418
 - testing (Apache Flex) 416
 - testing in Flex using automation support 259
 - testing in Flex using dynamic invoke 258
 - WPF 273
- custom exceptions
 - custom 525
- custom object
 - definition 573
- custom verification properties
 - defining 403
- Customer Care 21, 22
- customizing results 471
- CustomWin
 - large number of objects 297

D

- data member
 - definition 574
- data members
 - using properties instead 550
- data source
 - configuring DSN 189
- data sources
 - setting up 189
 - setting up for data-driven 189
- data-driven
 - workflow 184
- data-driven scripts
 - appending test cases, dialog box 198
 - creating, dialog box 198
 - overwriting, dialog box 198
 - specifying, dialog box 198
- data-driven test case
 - definition 574
- data-driven test cases
 - adding to test plans 195
 - automatically generated code 185
 - creating 190
 - data sources 189
 - overview 184
 - passing data to 194
 - running 193
 - running, using sample records 194
 - selecting test case 191
 - setting up data sources 189
 - specifying 199
 - tips and tricks 186
 - working with 185
- data-driven testing
 - finding values 191
 - replacing values 191
 - select test cases to data-drive 201
- data-driving test cases
 - Oracle 190
- databases
 - manipulating from test cases 223
 - testing 223
- DataTypeList
 - using 434
- DB Tester
 - using with Unicode content 423
- DBCS
 - definition 574
- debuggability
 - scripts 478
- debugger
 - about 478
 - executing scripts 479
 - exiting 480
 - menus 480
 - starting 479
- debugging
 - break statements, incorrect usage 486
 - changing variable values 483
 - designing debuggable scripts 478
 - enabling transcript 484
 - evaluating expressions 484
 - executing scripts 479
 - exit debugger 480
 - expressions 483
 - global variables, unexpected values 486
 - infinite loops 486
 - keyword-driven tests 478
 - loop variables, incorrect values 486
 - menus 480
 - never executing code 486
 - overview 478
 - setting breakpoints 481
 - starting debugger 479
 - step into 480
 - step over 480
 - test scripts 478, 479
 - tips 485
 - typographical errors 486
 - uninitialized handles 487
 - viewing modules 484
 - viewing transcripts 485
 - viewing variables 482
 - working with scripts 480
- debugging transcripts
 - viewing 485
- declarations
 - definition 574
 - dialog boxes 382
 - main window 384
 - menu 384
 - modified 421
 - overview 382
 - windows 385
- default agent
 - connecting 46
 - setting 45
 - setting, Runtime Options 45
 - setting, toolbar 46
- default browser
 - specifying 94
- default error handling 523
- DefaultBaseState
 - adding tests that use Open Agent 101
 - definition 574
 - function 101
 - wDynamicMainWindow object 104
- defaults.inc
 - overview 230
- DefaultScriptEnter method
 - overriding 106
- DefaultScriptExit method
 - overriding 106
- DefaultTestCaseEnter method
 - overriding 106
- DefaultTestCaseExit method
 - overriding 106
- DefaultTestPlanEnter method
 - overriding 106
- DefaultTestPlanExit method
 - overriding 106
- Define Attributes
 - dialog box 141

- define values
 - attributes 133
- defining
 - custom verification properties 403
 - exceptions, custom 525
- defining a custom verification property
 - example 404
- defining attributes
 - classes 399
 - with values 133
- defining classes
 - Open Agent 395
- defining custom verification properties
 - overview 400
- defining method example
 - adding method to TextField class 403
- defining methods
 - examples 403
 - overview 400
 - single GUI objects 401
- defining properties
 - classes 397
- defining symbols
 - Testplan detail dialog box 130
- Delete Breakpoint dialog box 482
- Delete Results
 - dialog box 471
- deleting
 - keywords 370
 - results 471
- deriving methods
 - from existing methods 402
- designing and recording test cases
 - test cases 172
- DesktopWin class
 - using 397
- determining where values are defined
 - large test plans 118
- device not connected
 - mobile 317
- DHTML
 - manually creating tests popup menus 353
- diacritic
 - definition 574
- Dialog
 - not recognized 356
- dialog box declarations
 - overview 382
- dialog boxes
 - Convert Results to Plan 114
 - declarations 382
 - Define Attributes 141
 - displaying double-byte characters 427
 - Edit Testplan Query 138
 - Find/Replace Values 191
 - Go to Testcase 201
 - Mark By Named Query 128
 - Mark by Query 126
 - New Testplan Query 138
 - Record Application State 150
 - Record Testcase 156
 - Run Testcase 447
 - Select Data Source 197
 - Select Testcase 201
 - Specify Data Driven Script 198
 - Specify Data Driven Testcase 199
 - specifying how to invoke 388
 - Testplan Completion Report 121
 - Testplan Detail 140
 - Testplan Detail - Testcase 141
 - Update Manual Tests 143
- DialogBox class
 - adding Tab method example 404
- Difference Viewer
 - about 452
 - definition 574
 - dialog box 452
- differences
 - agent options 56
 - agents, object recognition 57
 - moving to next or previous 470
- disabling extensions
 - browser 91
- display issues
 - Unicode content 428
- distributed testing
 - client/server testing configurations 205
 - connection, Open Agent 203
 - Open Agent 203
 - parallel processing 209
 - reporting distributed results 220
 - running tests on one remote target 217
 - running tests serially on multiple targets 218
 - specifying target machine driven by a thread 219
 - statement types 215
 - supported networking protocols, Open Agent 204
 - troubleshooting 232
 - using templates 216
- dividing test plans
 - master plan and sub-plans 118
- DLL calling conventions
 - stdcall 390
- dlls
 - aliasing names 390
 - calling from within 4Test scripts 390
 - common problems 543
 - definition 575
 - passing arguments to functions 391
 - using support files 393
- DLLs
 - calling 390
- do...except
 - statements 412
- do...except statements
 - handling exceptions 526
- do...except to handle exceptions 197
- Document Object Model
 - about 331
- documenting manual tests
 - test plans 116
- documenting user-defined methods
 - examples 437
- DOM
 - about 331

- DOM Extensions
 - dialog box 512
- double-byte character set
 - definition 574
- double-byte characters
 - displaying 427
 - displaying in dialog boxes 427
 - displaying in the Editor 428
 - issues 423
- double-byte files
 - reusing single-byte 424
- downloads 21, 22
- DSN
 - configuring for data-driven test cases 189
- duplicate test descriptions dialog box 144
- Dynamic HTML
 - manually creating tests for popup menus 353
- dynamic instantiation
 - definition 574
 - recording without window declarations 176
- dynamic invoke
 - FAQs 416
 - overview 415
 - simplify scripts 416
 - unexpected return value 416
- dynamic link library
 - definition 575
- dynamic object recognition
 - basic XPath concepts 159
 - locator keyword 169
 - overview 157
 - supported attribute types 162
 - supported XPath subset 159
 - XPath 158
- dynamically invoking methods
 - Flex 236
 - SAP 325
 - Silverlight 278
 - Windows Forms 268
 - Windows Presentation Foundation (WPF) 274
- DynamicInvoke
 - Apache Flex custom controls 258
 - Flex 236
 - Java AWT 284, 299
 - Java Swing 284, 299
 - Java SWT 284, 299
 - SAP 325
 - Silverlight 278
 - Windows Forms 268
 - Windows Presentation Foundation (WPF) 274
- DynamicInvokeMethods
 - Silverlight 278

E

- Eclipse
 - troubleshooting 300
- Edge
 - connection string 341
 - limitations 351
 - locators 355
 - recording tests 147
 - remote testing 352
 - testing 351
- Edit Property Set
 - dialog box 511
- Edit Testplan Query
 - dialog box 138
- editing
 - options sets 71
 - remote locations 155
- editor colors
 - setting 490
- Editor Colors
 - dialog box 490
- editor font
 - setting 490
- Editor Font
 - dialog box 490
- elapsed time
 - logging 475
- embedded browser applications
 - enabling extensions (Classic Agent) 88
- Emulator
 - defining, playback 312
- emulators
 - testing 301
- Enable Extensions dialog box
 - Classic Agent 94
- enabling
 - definition 575
- enabling extensions
 - automatically using basic workflow 86
 - host machines, manually 86
 - manually on target machines 87
- entering testdata statement
 - manually 123
- error handling
 - custom 523
 - default 523
- error messages
 - agent not responding 520
 - handling differences 405
 - troubleshooting 519
- error-handling
 - writing a function 529
- errors
 - handling 523
 - navigating to 473
- errors and the results file 450
- Euro symbol
 - displaying 545
- examples
 - adding a method to TextField class 403
 - adding Tab method to DialogBox class 404
- exception
 - definition 575
 - handling using do...except 197
- exception handling
 - do.. except statements 526
- exception values
 - Apache Flex 238
 - errors 530
- exceptions

- defining custom 525
- excluded characters
 - recording 202
 - replay 202
- executables
 - GUI-specific 413
- executing keyword-driven tests
 - variables 374
- executing scripts
 - debugging 479
- executing tests
 - Run Testcase dialog box 447
- existing files with Unicode content
 - specifying file formats 424
- existing tests
 - adding to projects 71
- Export Results
 - dialog box 455, 474
- exporting results
 - structured files 474
- Expression
 - dialog box 484
- expressions
 - about 483
 - evaluating, debugging 484
 - operator precedence 485
- extending class hierarchy
 - overview 393
- Extension Application
 - dialog box 513
- Extension Details
 - dialog box 505
- extension dialog boxes
 - adding test applications 89
- Extension Enabler
 - deleting applications 91
- Extension Enabler dialog box
 - comparison with Extensions dialog box 91
- Extension Information dialog box 95
- Extension Options dialog box
 - ActiveX 514
 - Java 514
- Extension Settings dialog box
 - client/server 96
 - Java 96
 - Web 95
- Extension Settings dialog box (.NET) 95
- extensions
 - automatically configurable 84
 - disabling 91
 - DOM Extensions dialog box 512
 - Enable Extensions dialog box (Classic Agent) 94
 - enabling automatically using basic workflow 86
 - enabling for AUTs 84
 - enabling for HTML applications 88
 - enabling manually on target machines 87
 - Extension Application dialog box 513
 - Extension Information dialog box 95
 - Extension Options dialog box (ActiveX) 514
 - Extension Options dialog box (Java) 514
 - Extension Settings dialog box 96
 - Extension Settings dialog box (.NET) 95

- Extension Settings dialog box (Java) 96
- Extension Settings dialog box (Web) 95
- Google Chrome 346
- host machines 85
- Mozilla Firefox 349
- overview 84
- set manually 85
- target machines 85
- verifying settings 90
- Extensions
 - deleting applications 91
 - dialog box 503
- Extensions dialog box
 - comparison with Extension Enabler dialog box 91
- Extract Results
 - dialog box 474
- extracting
 - results 474

F

- FAQs
 - cross-browser testing 353
 - deciding between 4Test methods and native methods 297
 - disabling CLASSPATH 297
 - dynamic invoke 416
 - invoking Java code 296
 - Java 296
 - many Java CustomWin objects 297
 - recording AWT menus 296
 - testing JavaScript objects 296
 - using Java plug-in outside JVM 296
- file
 - creating 567
 - frame 575
 - include 581
- file format issues
 - Unicode content 429
- file formats
 - about 423
 - existing files with Unicode content 424
 - new files with Unicode content 425
- file types
 - Silk Test Classic 72
- files
 - adding to projects 73
 - moving in a project 74
 - removing from projects 76
 - using 567
- files not displayed
 - recent files 82, 552
- files not found
 - projects 81, 551
- filtering
 - custom classes 421
 - keywords 377
- Find
 - dialog box 567
- Find dialog
 - example test cases 554
- Find/Replace Values

- dialog box 191
 - finding and replacing values
 - rules and tips 192
 - finding values
 - data-driven tests 191
 - Firefox
 - changing browser type for replay 330
 - configuration settings 340
 - cross-browser scripts 354
 - extensions, testing 349
 - limitations 350
 - locators 355
 - profiles, testing 349
 - testing 348
 - firewalls
 - port numbers 49
 - Flash player
 - opening applications in 234
 - security settings 234
 - Flex
 - adding configuration information 243
 - Adobe Air support 238
 - attributes 236
 - automation support for custom controls 260
 - automationIndex property 245
 - automationName property 245
 - class definition file 262
 - Component Explorer 250
 - Component Explorer, launching 250
 - containers 248
 - creating applications 244
 - custom controls 416
 - customizing scripts 235, 254
 - defining custom controls 255
 - enabling your application 240
 - exception values 238
 - Flash player settings 234
 - FlexDataGrid control, selecting items 239
 - initializing, applications 249
 - linking automation packages 240
 - loading at run-time 242
 - multiview containers 248
 - overview 233
 - passing parameters 243
 - passing parameters at runtime 243
 - passing parameters before runtime 243
 - precompiling the application 241
 - prerequisites 233, 250
 - recording, applications 249
 - run-time loading 242
 - sample test case 252
 - security settings 234
 - select method, overview 238
 - select method, setting 247
 - styles 235
 - testing 232
 - testing custom controls 254
 - testing custom controls using automation support 259
 - testing custom controls using dynamic invoke 258
 - testing multiple applications, same web page 237
 - testing playback 250
 - using dynamic invoke 236
 - verifying scripts 235, 254
 - workflow 249
 - FlexDataGrid control
 - selecting items 239
 - folders
 - adding to projects 74
 - available controls 74
 - moving in a project 74
 - removing from projects 75
 - renaming in projects 75
 - using in projects 74
 - fonts
 - displaying differently 429
 - setting 490
 - forward case-sensitive search
 - setup example 194
 - frame file
 - definition 575
 - frequently asked questions
 - cross-browser testing 353
 - deciding between 4Test methods and native methods 297
 - disabling CLASSPATH 297
 - dynamic invoke 416
 - invoking Java code 296
 - Java 296
 - recording AWT menus 296
 - testing JavaScript objects 296
 - to many Java CustomWin objects 297
 - using Java plug-in outside JVM 296
 - Frequently Asked Questions
 - AutoComplete 432
 - Fully customize a chart 475
 - fully qualified object name
 - definition 575
 - functional test design
 - incremental 265
 - functional testing
 - overview 266
 - functionality not supported
 - Open Agent 520
 - FunctionTip
 - using 434
- ## G
- General Options
 - dialog box 488
 - general protection faults
 - troubleshooting 546
 - generating completion reports
 - test plans 120
 - generating pass/fail reports
 - test plan results file 476
 - GetMachineData
 - multi-application testing example 226
 - GetProperty method
 - Flex 236
 - Java 284, 299
 - Silverlight 278
 - GetText
 - code sample 420

- getting started
 - Silk Test Classic 19
- global variables
 - GUI specifiers 411
 - local variables, same name 486
 - overview 210
 - protecting access 212
 - running from test plan versus running from script 547
 - unexpected values 486
 - viewing, debugging 482
- globalization
 - definition 581
- Globals dialog box 483
- glossary
 - overview 571
- Go to Testcase
 - dialog box 201
- Google Chrome
 - changing browser type for replay 330
 - configuration settings 340
 - extensions, testing 346
 - limitations 347
 - prerequisites 346
 - support 326
 - testing 345
 - user data directories, testing 347
- graphical controls
 - support 415
- group description
 - definition 575
- groups
 - sharing projects 68
- GUI objects
 - hierarchy 576
 - recording methods 402
- GUI specifiers
 - 4Test code 411
 - global variables 411
 - inheritance 410
 - overview 382, 409
 - syntax 411
 - usages 412
- GUI-specific captions
 - support 413
- GUI-specific executables
 - supporting 413
- GUI-specific menu hierarchies
 - support 413
- GUI-specific objects
 - support 412
- GWT
 - locating controls 165

H

- handles
 - definition 576
- handling GUI differences
 - porting tests 405
- hidden
 - input fields 353
- hidecalls

- keyword 400
- hierarchy of GUI objects
 - definition 576
- host machine
 - definition 576
- host machines
 - enabling extensions, manually 86
- hotkey
 - definition 576
- HTML applications
 - enabling extensions 88
- HTML definitions
 - tables, Open Agent 332
- Hungarian notation
 - definition 580
- hybrid applications
 - Android 302
 - iOS 308

I

- identifier
 - definition 580
- identifiers
 - overview 387
 - stable 163
- ignoring
 - classes, recording 152, 488, 505
- images
 - testing in Web applications 333
- IME
 - using 428
- IME issues
 - Unicode content 430
- IMEs
 - differing in appearance 430
- implementing
 - keywords 370
- improving
 - window declarations 386
- improving object recognition
 - Accessibility 389
- improving recognition
 - defining new window 386
- include file
 - definition 581
- include files
 - changes not applied 547
 - conditionally loading 405
 - handling very large files 232
 - loading for different test application versions 405
 - maximum size 232
- include scripts
 - changes not applied 547
- incorrect values
 - scripts, fixing 472
- incremental test design
 - functional 265
- indexing
 - schemes for 4Test and native Java methods 290
- infinite loops
 - debugging 486

- information service
 - configuring ports, clients 49
 - Mac, installing 309, 310, 342, 344
- inheritance
 - GUI specifiers 410
- initializing
 - Apache Flex applications 249
- innerHTML
 - xBrowser 353
- innerText
 - xBrowserf 353
- input fields
 - finding 353
- Input Method Editor
 - setting up 427
- Input Method Editor issues
 - Unicode content 430
- Input Method editors
 - differing in appearance 430
- Input Method Editors
 - using 428
- Insert Testplan Template dialog box 118
- installed apps
 - Android, testing 311
 - iOS, testing 311
- installing
 - information service, Mac 309, 310, 342, 344
 - privileges required 546
- installing language support
 - Unicode content 426
- installing USB drivers
 - Android 302
- insurance company sample Web application
 - testing 357
- insurance company Web application
 - configuring 358
 - creating a new project 357
 - recording test cases for Web site 358
 - replaying test cases 359
- integrations
 - configuring Silk Central location 375
- internationalization
 - configuring environment 426
 - definition 581
 - useful sites 423
- internationalized content
 - issues with displaying 423
- internationalized objects
 - support 422
- Internet Explorer
 - configuration settings 340
 - cross-browser scripts 354
 - link.select focus issue 355
 - locators 355
 - misplaced rectangles 355
 - support 326
- Internet Explorer 10
 - unexpected Click behavior 357
- invalid data
 - testing applications 188
- invalidated-handle error
 - troubleshooting 356

- invisible containers
 - about 421
- invoke
 - SAP 325
 - Windows Forms 268
 - Windows Presentation Foundation (WPF) 274
- Invoke method
 - callable methods 416
- invokeMethods
 - drawing line in multiline text field 294
- InvokeMethods
 - SAP 325
 - Windows Forms 268
 - Windows Presentation Foundation (WPF) 274
- invoking
 - dialog boxes 388
 - Java applications and applets 293
 - Java code from 4Test scripts 296
- invoking applets
 - Java 293
- invoking applications
 - JRE 294
 - JRE using -classpath 294
- invoking test cases
 - multi-application environments 224
- iOS
 - apps, preparing for testing 308
 - devices, preparing 308
 - hybrid applications 308
 - information service, installing 309, 310, 342, 344
 - installed apps, testing 311
 - mobile native applications, prerequisites 305
 - mobile web applications, prerequisites 305
 - multiple devices 310
 - multiple Simulators 310
 - native app, Simulator 306
 - native app, testing 305
 - recommended settings 310
 - recording test cases 53, 146
 - releasing devices 316
 - releasing devices, recording 316
 - releasing devices, replay 316
 - testing 304
 - web app, Simulator 307
 - web app, testing 307

J

- Java
 - accessing objects and methods 295
 - applet controls not recognized 291, 534
 - calling nested native methods 295
 - disabling CLASSPATH 297
 - enabling support 287
 - FAQs 296
 - invoking applets 293
 - invoking from 4Test scripts 296
 - javaex.inc 291
 - predefined class definition file 291
 - security privileges 289
 - testing scroll panes 296
- Java applets

- invoking 293
 - supported browsers 285
 - Java Applets
 - configuring 288
 - Java applications
 - Silk Test Java file missing in plug-in 291, 534
 - configuring standalone applications 288
 - prerequisites 287
 - standard names 90
 - troubleshooting 291, 534
 - Java applications and applets
 - invoking 293
 - preparing for testing 290
 - testing 290
 - Java AWT
 - agent support 282
 - dynamically invoking methods 284, 299
 - DynamicInvoke 284, 299
 - locator attributes 284
 - object recognition 282
 - Open Agent 281
 - supported controls 282
 - Java AWT menus
 - playing back 282
 - recording 282
 - Java AWT/Swing
 - priorLabel 285
 - testing standard Java objects 281
 - Java database connectivity
 - definition 581
 - Java Development Kit
 - definition 581
 - Java extension
 - enabling 287
 - Java FAQs
 - overview 296
 - Java Foundation Class
 - playing back menus 282
 - recording menus 282
 - Java Foundation Classes
 - definition 581
 - Java Network Launching Protocol
 - configuring test applications 283
 - Java objects
 - accessing nested 295
 - Java plug-in
 - using outside JVM 296
 - Java Runtime Environment
 - definition 581
 - Java scroll panes
 - testing 296
 - Java security policy
 - changing 289
 - Java security privileges
 - changing 289
 - Java support
 - enabling 287
 - manually configuring for Sun JDK 288
 - Sun JDK 288
 - supported classes 291
 - Java Swing
 - dynamically invoking methods 284, 299
 - DynamicInvoke 284, 299
 - Java SWT
 - locator attributes 299
 - dynamically invoking methods 284, 299
 - DynamicInvoke 284, 299
 - troubleshooting 300
 - Java SWT and Eclipse
 - Open Agent 297
 - Java Virtual Machine
 - definition 581
 - Java-equivalent window classes
 - predefined 292
 - JavaBeans
 - definition 581
 - JavaScript
 - support 285
 - testing 296
 - JBuilder
 - support 296
 - JDBC
 - definition 581
 - JDeveloper
 - support 296
 - JDK
 - definition 581
 - JFC
 - definition 581
 - playing back menus 282
 - recording menus 282
 - JFC classes
 - predefined 292
 - JNLP
 - configuring test applications 283
 - JRE
 - definition 581
 - invoking applications 294
 - invoking applications using -classpath 294
 - JVM
 - definition 581
- ## K
- keyword libraries
 - uploading 376
 - keyword sequences
 - creating 374
 - parameters 372
 - keyword-driven
 - testing 364
 - keyword-driven test editor
 - recommended keywords 371
 - keyword-driven testing
 - advantages 364
 - keyword recommendations, algorithm 371
 - marking test methods 368
 - overview 364
 - parameters, example 372
 - keyword-driven tests
 - adding keywords 369
 - application configurations 367
 - base state 367
 - creating 366

- debugging 478
- editing 369
- implementing keywords 368
- implementing Silk Central keywords 375
- keywords, searching 377
- recording 366
- removing keywords 369
- specifying variables, execution 374
- uploading keywords, Silk Central 376
- keywords
 - about 365
 - adding 370
 - combining 370, 374
 - deleting 370
 - filtering 377
 - finding in project 377
 - hidecalls 400
 - implementing 368, 370
 - locator 169
 - managing 370
 - marking test methods 368
 - nesting 370
 - opening 370
 - parameters 370, 372
 - parameters, example 372
 - recording 368
 - replacing 370
 - sequences 370
 - uploading to Silk Central 376

L

- Language bar
 - only English listed 430
- large test plans
 - determining where values are defined 118
 - overview 118
- Latin script
 - definition 582
- layout
 - definition 582
- libraries
 - uploading 376
- Library Browser
 - adding information 436
 - adding user-defined files 437
 - not displaying user-defined methods 548
 - not-displayed Web classes 438
 - overview 435
 - source file 435
 - viewing class methods 437
 - viewing functions 437
- licenses
 - handling limited licenses 232
- licensing
 - available license types 18
- limitations
 - Apple Safari 343
 - Google Chrome 347
 - Microsoft Edge 351
 - mobile web applications 320
 - Mozilla Firefox 350

- native mobile applications 321
- Windows 10 363
- linking descriptions to scripts
 - Testplan Details dialog box 123
- linking descriptions to test cases
 - Testplan Details dialog box 123
- linking test plans to test cases
 - example 125
- links
 - testing 333
- load testing
 - definition 582
- loading include files
 - conditionally 405
- local sub-plan copies
 - refreshing 119
- local variables
 - global variables, same name 486
 - viewing, debugging 482
- localization
 - definition 582
- localization levels
 - definition 582
- localizing applications
 - definition 582
- locally testing multiple applications
 - sample include file (Classic Agent) 536
 - sample script file (Classic Agent) 535
- Locals dialog box 483
- locating controls
 - GWT example 165
 - siblings example 164
- locator
 - definition 582
 - keyword 169
- locator attributes
 - Apache Flex controls 236
 - excluded characters 202
 - Java AWT applications 284
 - Java SWT 299
 - Rumba controls 324
 - SAP 325
 - Silverlight controls 277
 - Swing applications 284
 - Windows API-based controls 360
 - Windows Forms controls 267
 - WPF controls 271
 - xBrowser controls 335
- locator generator
 - configuring for xBrowser 338
- locator keyword
 - overview 169
- locator keywords
 - recording window declarations 148
- locator recognition
 - enhancing 553
- Locator Spy
 - dialog box 149, 156
 - recording locators 149
- locators
 - customizing 163
 - incorrect in xBrowser 355

- object types 159
- recording using Locator Spy 149
- search scopes 159
- setting custom attributes 508
- supported subset 162
- using attributes 159
- xBrowser 355
- locks
 - acquiring 120
 - overview 120
 - releasing 120
 - test plans 120
- logging
 - elapsed time 475
 - machine information 475
 - thread 475
- logging errors
 - programmatically 526
- logic errors
 - evaluating 452
- logical controls
 - different implementations 406
- logical hierarchy
 - definition 583
- login windows
 - handling 107
 - non-Web applications (Open Agent) 107
- looking at statistics
 - bitmaps 465
- loop variables
 - incorrect values 486
- IsLeaveOpenLocators
 - specifying windows to be left open (Open Agent) 108
- lwLeaveOpenWindows
 - specifying windows to be left open (Open Agent) 108

M

- Mac
 - Apple Safari, prerequisites 342
 - Apple Safari, testing 342
 - information service, installing 309, 310, 342, 344
- machine handle operator
 - specifying 219
- machine handle operators
 - alternative syntax 220
- machine information
 - logging 475
- main function
 - using in scripts 196
- main window
 - declarations 384
- managing
 - keywords 370
- manual test
 - definition 583
 - describing the state 116
- manual test state
 - describing 116
- manual tests
 - updating 143
- mark

- definition 583
- Mark by Named Query
 - dialog box 128
- Mark by Query
 - dialog box 126
- marked tests
 - printing 126
- marking commands
 - interactions 126
- masks
 - applying 467
 - creating one that excludes all differences 468
 - editing 467
 - excluding differences 467
 - excluding selected areas 467
 - prerequisites 467
 - saving 469
- master plan
 - definition 583
- master plans
 - connecting with sub-plans 119
- maximum size
 - Silk Test Classic files 548
- MBCS
 - definition 584
- member-of operator
 - using to access data 387
- MemberList
 - customizing 431
 - using 434
- menu
 - declarations 384
- menu hierarchies
 - GUI-specific 413
- Merge Results
 - dialog box 452, 472
- merging
 - results, test plans 452, 472
- message box
 - definition 583
- messages sent to the result file
 - relationship with exceptions defined in 4test.inc 549
- method
 - definition 583
- methods
 - adding to existing classes 400
 - adding to single GUI objects 401
 - agent support 64
 - defining 400
 - defining for single GUI objects 401
 - deriving new from existing 402
 - recording for GUI objects 402
 - redefining 403
- Microsoft Accessibility
 - improving object recognition 389
- Microsoft Edge
 - connection string 341
 - limitations 351
 - recording tests 147
 - remote testing 352
 - support 326
 - testing 351

- migrating
 - from the Classic Agent to the Open Agent 56
- minus (-) sign
 - definition 583
- missing peripherals
 - test machines 19
- mobile
 - troubleshooting 317
- mobile applications
 - recording 311
 - recording test cases 53, 146
 - testing 300
- mobile browsers
 - limitations 320
- mobile device
 - defining, playback 312
- mobile devices
 - interacting with 315
 - performing actions against 315
- mobile native applications
 - limitations 321
- mobile recording
 - about 311
- mobile testing
 - Android 301
 - connection string 312
 - iOS 304
 - native app, iOS Simulator 306
 - overview 300
 - releasing devices 316
 - remote locations, adding 155
 - replay, parallel 446
 - web app, iOS 307
 - web app, iOS Simulator 307
- mobile testing devices
 - native app, iOS 305
- mobile web
 - Click 322
 - iOS 307
 - legacy tests 323
- mobile web applications
 - Android, prerequisites 301
 - iOS, prerequisites 305
 - limitations 320
- mobile Web applications
 - configuring 251
- modal
 - definition 583
- modeless
 - definition 584
- modified declarations
 - using 421
- modifying identifiers
 - test frames 329
- modules
 - viewing, debugging 484
- MoveableWin
 - cannot extend class 542
- moving files
 - between projects 75
 - on Files tab 74
- moving folders
 - in a project 74
- Mozilla Firefox
 - changing browser type for replay 330
 - configuration settings 340
 - extensions, testing 349
 - limitations 350
 - profiles, testing 349
 - support 326
 - supported versions, new 25
 - testing 348
- multi-application environments
 - cs.inc 231
- multi-application testing
 - code for template.t 229
 - invoking example 229
 - invoking example explained 229
 - invoking test cases 224
 - overview 223
 - template.t explained 229
- multi-test case
 - statements 224
- multibyte character set
 - definition 584
- multiple agents
 - single machine 441
- Multiple Application Domains (.NET)
 - definition 584
- multiple applications
 - setting up the recovery system 534
 - single machine 441
- multiple devices
 - iOS 310
- multiple Flex applications
 - testing, same web page 237
- multiple machines
 - driving 211
 - testing, Open Agent 203
 - troubleshooting 534
- multiple Simulators
 - iOS 310
- multiple tests
 - recovering 210
- multiple verifications
 - test cases 527
- multiple-application environments
 - test case structure 224

N

- Named Query command
 - differences with Query 136
- native Java controls
 - versus 4Test 290
- native Java methods
 - comparing with 4Test methods 297
- native mobile applications
 - Android, prerequisites 301
 - iOS, prerequisites 305
 - limitations 321
- native playback
 - comparing to, API playback 339
- native user input

- advantages 339
- navigating to errors 473
- negative testing
 - definition 584
- nested declarations
 - definition 584
- nested Java objects
 - accessing 295
- nesting
 - keywords 370
- network
 - configuring 204
- network testing
 - types of testing 265
- networking
 - supported protocols, Open Agent 204
- networks
 - enabling on remote host 205
- never executing code
 - checking 486
- New Attribute dialog box 142
- new files with Unicode content
 - specifying file formats 425
- new projects
 - creating 51, 69, 251
- New Property Set
 - dialog box 511
- New Testplan Query
 - dialog box 138
- no-touch (.NET)
 - definition 584
- non-Web applications
 - handling login windows (Open Agent) 107
- not all actions captured
 - recorder 549
- NumChildren
 - alternative class property 400

O

- object
 - definition 584
- object files
 - advantages 381
 - locations 381
 - overview 380
- object properties
 - overview 179
 - verifying 179
 - verifying (Open Agent) 179
- object recognition
 - agent differences 57
 - creating stable locators 163
 - dynamic 157
 - Exists method 162
 - FindAll method 163
 - identifying multiple objects 163
 - improving by defining new window 386
 - improving with Accessibility 389
 - Java AWT 282
 - objects recognized as custom controls 414
 - Swing 282

- using attributes 159
- object types
 - locators 159
- object-oriented programming languages
 - classes 287
- objects
 - checking for existence 162
 - internationalized 422
 - properties 179
 - verifying properties 179
 - verifying properties (Open Agent) 179
 - verifying state 181
- objects recognized as custom controls
 - reasons 414
- Open Agent
 - adding tests to the DefaultBaseState 101
 - comparison to Classic Agent, classes 59
 - configuring port numbers 48, 205
 - configuring ports, information service 49
 - migrating to from Classic Agent 56
 - overview 26
 - port numbers 49
 - recording options, setting 152, 488, 505
 - recording test cases 52, 145
 - replay options, setting 446, 508
 - setting the recovery system 99
 - starting from script 50
 - stopping from script 50
 - testing, remote 203
- Open Agent parameters
 - comparison to Classic Agent 63
- opening
 - keywords 370
 - TrueLog Options dialog box 458
- opening projects
 - existing 70
- operators
 - checking precedence 485
- OPT_AGENT_CLICKS_ONLY
 - option 26
- OPT_ALTERNATE_RECORD_BREAK
 - option 27
- OPT_APPREADY_RETRY
 - option 27
- OPT_APPREADY_TIMEOUT
 - option 27
- OPT_BITMAP_MATCH_COUNT
 - option 27
- OPT_BITMAP_MATCH_INTERVAL
 - option 28
- OPT_BITMAP_MATCH_TIMEOUT
 - option 28
- OPT_BITMAP_PIXEL_TOLERANCE
 - option 29
- OPT_CLASS_MAP
 - option 29
- OPT_CLOSE_CONFIRM_BUTTONS
 - option 29
- OPT_CLOSE_DIALOG_KEYS
 - option 29
- OPT_CLOSE_MENU_NAME
 - option 29

OPT_CLOSE_WINDOW_BUTTONS
 option 29
 OPT_CLOSE_WINDOW_MENUS
 option 30
 OPT_CLOSE_WINDOW_TIMEOUT
 option 30
 OPT_COMPATIBILITY
 option 30
 OPT_COMPATIBLE_TAGS
 option 30
 OPT_COMPRESS_WHITESPACE
 option 30
 OPT_DROPDOWN_PICK_BEFORE_GET
 option 31
 OPT_ENABLE_ACCESSIBILITY
 option 31
 OPT_ENSURE_ACTIVE_WINDOW
 option 32
 OPT_EXTENSIONS
 option 32
 OPT_GET_MULTITEXT_KEEP_EMPTY_LINES
 option 32
 OPT_ITEM_RECORD
 option 32
 OPT_KEYBOARD_DELAY
 option 32
 OPT_KEYBOARD_LAYOUT
 option 32
 OPT_KILL_HANGING_APPS
 option 33
 OPT_LOCATOR_ATTRIBUTES_CASE_SENSITIVE
 option 33
 OPT_MATCH_ITEM_CASE
 option 33
 OPT_MENU_INVOKE_POPUP
 option 33
 OPT_MENU_PICK_BEFORE_GET
 option 33
 OPT_MOUSE_DELAY
 option 34
 OPT_MULTIPLE_TAGS
 option 34
 OPT_NO_ICONIC_MESSAGE_BOXES
 option 34
 OPT_PAUSE_TRUELOG
 option 34
 OPT_PLAY_MODE
 option 34
 OPT_POST_REPLAY_DELAY
 option 35
 OPT_RADIO_LIST
 option 35
 OPT_RECORD_LISTVIEW_SELECT_BY_TYPEKEYS
 option 35
 OPT_RECORD_MOUSE_CLICK_RADIUS
 option 35
 OPT_RECORD_MOUSEMOVES
 option 35
 OPT_RECORD_SCROLLBAR_ABSOLUT
 option 35
 OPT_REL1_CLASS_LIBRARY
 option 36
 OPT_REMOVE_FOCUS_ON_CAPTURE_TEXT
 option 36
 OPT_REPLAY_HIGHLIGHT_TIME
 option 36
 OPT_REPLAY_MODE
 option 36
 OPT_REQUIRE_ACTIVE
 option 36
 OPT_RESIZE_APPLICATION_BEFORE_RECORDING
 option 37
 OPT_SCROLL_INTO_VIEW
 option 37
 OPT_SET_TARGET_MACHINE
 option 37
 OPT_SHOW_OUT_OF_VIEW
 option 37
 OPT_SYNC_TIMEOUT
 option 38
 OPT_TEXT_NEW_LINE
 option 38
 OPT_TRANSLATE_TABLE
 option 38
 OPT_TRIM_ITEM_SPACE
 option 38
 OPT_USE_ANSICALL
 option 38
 OPT_USE_SILKBEAN
 option 38
 OPT_VERIFY_ACTIVE
 option 38
 OPT_VERIFY_APPREADY
 option 39
 OPT_VERIFY_CLOSED
 option 39
 OPT_VERIFY_COORD
 option 39
 OPT_VERIFY_CTRLTYPE
 option 39
 OPT_VERIFY_ENABLED
 option 40
 OPT_VERIFY_EXPOSED
 option 40
 OPT_VERIFY_RESPONDING
 option 40
 OPT_VERIFY_UNIQUE
 option 40
 OPT_WAIT_ACTIVE_WINDOW
 option 40
 OPT_WAIT_ACTIVE_WINDOW_RETRY
 option 41
 OPT_WINDOW_MOVE_TOLERANCE
 option 41
 OPT_WINDOW_RETRY
 option 41
 OPT_WINDOW_SIZE_TOLERANCE
 option 42
 OPT_WINDOW_TIMEOUT
 option 42
 OPT_WPF_CHECK_DISPATCHER_FOR_IDLE
 option 43
 OPT_WPF_CUSTOM_CLASSES
 option 43

- OPT_WPF_PREFILL_ITEMS
 - option 43
- OPT_XBROWSER_FIND_HIDDEN_INPUT_FIELDS
 - option 44
- OPT_XBROWSER_SYNC_EXCLUDE_URLS
 - option 44
- OPT_XBROWSER_SYNC_MODE
 - option 44
- OPT_XBROWSER_SYNC_TIMEOUT
 - option 44
- optimizing replay
 - replay options, setting 446, 508
- options
 - agents 26
 - recording, Classic Agent 507
 - recording, Open Agent 152, 488, 505
 - replay, Open Agent 446, 508
 - sets 406
 - sets, adding to projects 71
 - setting 488
- options sets
 - editing in projects 71
 - porting 406
 - specifying 406
 - using in projects 71
- Oracle DSN
 - data-driving test cases 190
- Oracle Forms
 - about 286
 - attributes 286
 - prerequisites 286
 - supported versions 286
- organizing
 - projects 73
- other options
 - Agent Options dialog box 501
- outline
 - definition 584
- overriding
 - default recovery system 106
- overwriting data-driven scripts
 - Specify Data Driven Script dialog box 198

P

- packaged projects
 - emailing 78
- packaging
 - projects 76
- page synchronization
 - xBrowser 336
- parallel processing
 - spawn statement 215
 - statements 215
- parallel replay
 - mobile tests 446
- parallel statements
 - using 215
- parallel test cases
 - using templates 216
- parallel testing
 - asynchronous 214
- parameters
 - agent comparison 63
 - handling, keywords 372
- parent object
 - definition 585
- pass/fail chart
 - creating 476
- passing arguments
 - scripts 443
 - to DLL functions 391
- passing data
 - data-driven test cases 194
- peak load testing 266
- performance testing
 - definition 585
- physical hierarchy (.NET)
 - definition 585
- playback
 - selecting device 312
- plus (+) sign
 - definition 585
- polymorphism
 - concept 394
 - definition 585
- popup menus
 - manually creating tests 353
- port numbers
 - configuring, Open Agent 48, 205
 - Open Agent 49
- porting tests
 - another GUI 405
 - differences between GUIs 405
- ports
 - configuring, information service 49
- pre-fill
 - setting during recording and replaying 276
- predefined attributes
 - test plan editor 132
- predefined classes
 - AWT 292
- prerequisites
 - Android, mobile web applications 301
 - Android, native mobile applications 301
 - Apple Safari 342
 - Flex 233, 250
 - Google Chrome 346
 - iOS, mobile web applications 305
 - iOS, native mobile applications 305
 - testing Java applications 287
- Print
 - dialog box 569
- Printer Setup
 - dialog box 569
- printers
 - setting up 569
- printing
 - active window contents 569
 - marked tests 126
- priorLabel
 - Java AWT/Swing technology domain 285
 - Win32 technology domain 363
 - Windows Forms 271

- privileges required
 - installing Silk Test 546
 - running Silk Test 546
- product suite
 - components 20
- Product Support 21, 22
- profiles
 - Mozilla Firefox 349
- project
 - definition 585
- Project Description
 - dialog box 83
- Project Explorer
 - overview 68
 - sorting resources 75
 - turning on and off 76
 - Unicode characters do not display 429
- project files
 - editing 82, 553
 - not loaded 81, 551
- project information
 - storing 66
- project-related information
 - storing 66
- projects
 - about 66
 - accessing files 67
 - adding an options set 71
 - adding existing tests 71
 - adding files 73
 - adding folders 74
 - cannot load project file 81, 551
 - cannot save files 81, 552
 - creating 51, 69, 251
 - editing project files 82, 553
 - editing the options set 71
 - emailing packaged projects 78
 - exporting 80
 - files not found 81, 551
 - moving files between 75
 - moving files in projects 74
 - moving folders in projects 74
 - opening existing projects 70
 - organizing 73
 - packaging 76
 - Project Description dialog box 83
 - removing files 76
 - removing folders 75
 - renaming 73
 - renaming folders 75
 - sharing among a group 68
 - storing information 66
 - troubleshooting 81, 551
 - turning Project Explorer on and off 76
 - viewing associated files 76
 - viewing resources 76
 - working with folders 74
- properties
 - definition 586
 - objects 179
 - using instead of data members 550
 - verifying 398

- property list
 - confirming 403
- property sets
 - Combine Property Sets dialog box 511
 - Edit Property Set dialog box 511
 - New Property Set dialog box 511
 - Property Sets dialog box 510
- Property Sets
 - dialog box 510
- protocols
 - networking, Open Agent 204

Q

- queries
 - building 195
 - combining 138
 - combining to create new 135
 - creating, named 128
 - creating, test plans 136
 - deleting 137
 - editing 137
 - including symbols 136
 - test plans 135
- query
 - definition 586
- Query command
 - differences with Named Query 136

R

- recent files
 - files not displayed 82, 552
- recognizing controls
 - as custom controls 414
- recognizing objects
 - xBrowser 334
- recommendations
 - algorithm 371
- recommended keywords
 - keyword-driven test editor 371
- Record Application State
 - dialog box 150
- record functionality
 - agents 47
- Record Method
 - dialog box 402
- record options
 - Classic Agent 507
 - Open Agent 152, 488, 505
- Record Testcase
 - dialog box 156
- recorder
 - adding properties 541
 - does not capture all actions 549
- Recorder Options
 - dialog box, Classic Agent 507
- recording
 - actions into existing tests 151
 - Apache Flex applications 249
 - application states 150
 - available actions 151

- available functionality 47
- AWT menus 296
- keyword-driven tests 366
- keywords 368
- locators using Locator Spy 149
- methods for GUI objects 402
- mobile applications 311
- object highlighting 169
- releasing devices 316
- remote 211
- resolving window declarations 154
- setting classes to ignore 509
- setting options, Classic Agent 507
- setting options, Open Agent 152, 488, 505
- setting pre-fill 276
- test cases 156
- test cases for mobile applications 53, 146
- test cases, Open Agent 52, 145
- test frames 380
- Update Files dialog box 157
- using locators or tags 154
- without window declarations 176
- recording a close method
 - Open Agent 109
- recording actions
 - existing tests 151
- recording options
 - setting for xBrowser 339
- Recording Options
 - dialog box, Open Agent 152, 488, 505
- recording test cases
 - insurance company Web site 358
 - mobile applications 53, 146
 - Open Agent 52, 145
- recording tests
 - Microsoft Edge 147
- recording window declarations
 - locator keywords 148
 - main window 387
 - menu hierarchy 387
- recovery system
 - closing windows 104
 - defaults.inc file 230
 - definition 586
 - flow of control 104
 - modifying 105
 - Open Agent 98
 - overriding default 106
 - setting for the Open Agent 99
 - specifying new window closing procedures 108
 - starting the application 105
- regression testing
 - definition 586
- releasing devices
 - mobile testing 316
 - recording 316
 - replay 316
- releasing locks
 - test plans 120
- remote agent
 - about 203
- remote agents
 - connecting, Open Agent 203
- remote applications
 - multiple 204
 - single 204
- remote browser testing
 - connection string 341
- remote locations
 - adding 155
 - editing 155
- remote testing
 - Microsoft Edge 352
 - Open Agent 203
- removing
 - unused space, result files 475
- removing keywords
 - keyword-driven tests 369
- renaming
 - projects 73
- Replace
 - dialog box 568
- replacing
 - keywords 370
 - strings 568
- replacing values
 - data-driven tests 191
- replay
 - Dialog not recognized 356
 - releasing devices 316
 - selecting device 312
 - setting options, Open Agent 446, 508
- replay options
 - Agent Options dialog box 446, 508
 - setting, Open Agent 446, 508
- replaying
 - setting classes to ignore 509
 - setting pre-fill 276
- replaying test cases
 - insurance company Web application 359
- replaying tests
 - remote machines 203
- reporting
 - distributed results 220
- reports
 - presenting results 475
- reraise statement
 - error handling 523
- resolving window declarations
 - using locators or tags 154
- result bitmaps
 - baseline bitmaps, graphical differences 469
- result files
 - changing the color of elements 472
 - comparing 470
 - using 470
- result sets
 - attaching comments 470
 - default number, changing 471
- results
 - converting to test plans 114
 - customize a chart 475
 - customizing 471
 - deleting 471

- displaying 454
- displaying a different set 477
- errors and results file 450
- exporting, structured files 474
- extracting 474
- interpreting 449
- marking failed test cases 472
- merging, test plans 452, 472
- presenting 475
- removing unused space, result files 475
- results file overview 449
- scripts, fixing incorrect values 472
- selecting 454
- starting the Bitmap Tool from the results file 466
- storing 473
- storing and exporting 473
- testplan pass-fail report and chart 452
- viewing an individual summary 473
- results file
 - definition 586
 - overview 449
- results files
 - not opening 543
 - test plans, converting 114
- Results Pass/Fail Report
 - dialog box 456
- rules and tips
 - finding and replacing values 192
- Rumba
 - about 323
 - enabling and disabling support 323
 - locator attributes 324
 - Unix display 324
- Rumba locator attributes
 - identifying controls 324
- Run Application State
 - dialog box 447
- Run Testcase
 - dialog box 447
- running
 - active script or suite 445
 - global variables, test plan versus script 547
 - test cases, data driven 193
 - test plans 445
 - tests 443
- running multiple tests
 - Apple Safari 344
- running test cases
 - stopping 447
- Runtime
 - about 556
 - comparing with Silk Test Classic 556
 - installing 556
 - starting 556
- Runtime Options
 - dialog box 491
- Runtime Status
 - dialog box 448

S

Safari

- connection string 341
- limitations 343
- prerequisites 342
- running multiple tests 344
- testing 342
- sample applications
 - Component Explorer, launching 250
- SAP
 - class reference 326
 - invoking methods 325
 - locator attributes 325
 - overview 324
 - testing 324
- saving captured bitmaps
 - Bitmap Tool 463
- saving changes
 - sub-plans 120
- saving existing files
 - Save as dialog box opens 430
- sBrowserType
 - browsers, switching 330
- script
 - definition 586
- script deadlocks
 - 4Test handling 266
- script file
 - definition 586
- script files
 - saving 154
- ScriptEnter method
 - overriding default recovery system 106
- ScriptExit method
 - overriding default recovery system 106
- scripting
 - common problems 544
- scripts
 - adding verifications while recording 179
 - creating, best practices 153
 - deadlock handling 266
 - incorrect values, fixing 472
 - marking tests as keywords 368
 - passing arguments to 443
 - saving 154
 - using main function 196
- search scopes
 - locators 159
- search setup example
 - forward case-sensitive search 194
- searching
 - keywords, keyword-driven tests 377
 - strings 567
- security privileges
 - Java 289
- Select Data Source
 - dialog box 197
- Select method
 - Apache Flex, setting 247
- Select Results
 - dialog box 454
- Select Results to Chart
 - dialog box 455
- Select Testcase

- dialog box 201
- selecting items
 - FlexDataGrid control 239
- selecting test cases
 - to data drive 191
- serial number 21, 22
- Set attributes
 - adding members 132
 - removing members 132
- Set Recovery System
 - dialog box 110
- SetProperty method
 - Flex 236
 - Java 284, 299
 - Silverlight 278
- SetText
 - code sample 420
- setting
 - options 488
- setting agent options
 - Web testing 93
- setting classes to ignore
 - transparent classes 509
- setting default agent
 - Runtime Options dialog box 45
 - toolbar 46
- setting mobile device
 - playback 312
- setting options
 - TrueLog 458
 - TrueLog Explorer 458
- setting recording options
 - xBrowser 339
- setting the recovery system
 - Open Agent 99
- setting up IME
 - Unicode content 427
- setting up the recovery system
 - multiple local applications 534
- Setup Data Driven Script DSN
 - dialog box 198
- shared data
 - specifying 122
- sharing initialization files
 - test plans 120
- sharing projects
 - groups 68
- siblings
 - locating 164
- side-by-side (.NET)
 - definition 586
- Silk Central
 - configuring location 375
 - setting browser, replay 330
 - uploading keywords 376
- Silk Central keywords
 - implementing 375
- Silk Test Classic
 - about 19
 - not starting 82, 552
- Silk Test Classic files
 - maximum size 548
- Silverlight
 - invoking methods 278
 - locator attributes 277
 - object recognition 277
 - scrolling 279
 - testing 277
 - troubleshooting 280
- Silverlight locator attributes
 - identifying controls 277
- Simplified Chinese
 - definition 587
- Simulator
 - defining, playback 312
 - mobile web applications, testing 307
 - native app, testing 306
 - testing 305
- single applications
 - remote 204
- single GUI objects
 - defining methods 401
- single-application environments
 - test case structure 224
- single-application tests
 - recovery-system file 230
- single-byte character set (SBCS) 587
- single-byte files
 - reusing as double-byte 424
- sleep
 - adding to tests 441
- slowing down
 - tests 441
- smoke test 587
- sorting resources
 - Project Explorer 75
- spawn
 - multi-application testing example 226
- spawn statement
 - using 215
- specifiers
 - GUI 382
- Specify Data Driven Script
 - dialog box 198
- Specify Data Driven Testcase
 - dialog box 199
- Specify Rows
 - dialog box 199
- specifying
 - target machine for a single command 219
- specifying browser
 - testing Web applications 93
- specifying data-driven scripts
 - Specify Data Driven Script dialog box 198
- specifying new window closing procedures
 - recovery system 108
- specifying windows to be left open
 - Open Agent 108
- stable identifiers
 - about 163
- stable locators
 - creating 164
- standard applications
 - configuring 362

- Standard Widget Toolkit (SWT) 587
- starting
 - command line 378
- starting Bitmap Tool
 - from icon 466
 - from the results file 466
- starting from the command line
 - Silk Test Classic 378
- starting Open Agent
 - scripts 50
- starting the Bitmap Tool
 - Run dialog box 466
- statement
 - definition 587
- statements
 - do...except 412
 - parallel 215
 - type 412
- status line 587
- stdcall
 - DLL calling conventions 390
- step into
 - debugging 480
- step over
 - debugging 480
- stopping
 - running test cases 447
- stopping Open Agent
 - scripts 50
- storing and exporting
 - results 473
- storing results 473
- str function
 - does not round correctly 551
- stress testing 587
- structured files
 - results, exporting 474
- sub-plans
 - connecting with master plans 119
 - copying 119
 - opening 119
 - refreshing local copies 119
 - saving changes 120
- subplan
 - definition 588
- suite
 - definition 588
- suites
 - creating 443
- Sun JDK
 - Java support 288
 - manually configuring Java support 288
- supported browsers
 - testing Java applets 285
- supported controls
 - Java AWT 282
 - Swing 282
- supported Java classes
 - overview 291
- SupportLine 21, 22
- suppressing controls
 - Classic Agent 361
 - Open Agent 270, 298, 361
- Swing
 - agent support 282
 - definition 588
 - locator attributes 284
 - object recognition 282
 - Open Agent 281
 - supported controls 282
- switching browsers
 - replay 330
- symbols
 - assigning values 131
 - definition 588
 - including in queries 136
 - overview 128
 - specifying as arguments for testcase statements 131
 - using 128
- symbolvalue
 - assigning to symbol 131
- synchronization
 - about 453
 - changing settings 453
 - wrong timestamps 355
 - xBrowser 336
- synchronization options
 - Agent Options dialog box 499
 - xBrowser, setting 337
- synchronizing threads with semaphores 212
- sys functions
 - agent specific differences 64
- system dialog boxes
 - cannot display multiple languages 428

T

- tables
 - building queries 195
 - definition, Open Agent 332
 - testing in Web applications 332
 - verifying in client/server applications 263
- tag
 - definition 588
- target machine
 - definition 588
- target machines
 - manually enabling extensions 87
- template
 - definition 589
- templates
 - test plans 112
- test application settings
 - copying 90
- test applications
 - adding to extension dialog boxes 89
 - deleting from Extension Enabler dialog box 91
 - deleting from Extensions dialog box 91
 - duplicating settings 90
 - loading different include files for different application versions 405
- test automation
 - obstacles 19
 - synchronization 453

- test case
 - definition 589
- test case example
 - word processor feature 177
- test case structure
 - multiple-application environments 224
 - single-application environments 224
- test cases
 - about 172
 - anatomy of basic test case 173
 - constructing 174
 - creating (Open Agent) 145
 - data 175
 - data-driven 184
 - designing 173
 - designing and recording, Open Agent 145
 - example word processor feature 177
 - marking failed 472
 - overview 172
 - recording 156
 - running 54, 253, 444
 - running data driven 193
 - running in animation mode 447
 - saving 176
 - stopping during execution 447
 - types 173
 - with multiple verifications 527
- test description
 - definition 589
- test frame file 589
- test frames
 - modifying identifiers 329
 - overview 328
 - recording 380
 - recording, web applications (Open Agent) 329
 - saving 388
- test machines
 - missing peripherals 19
- test methods
 - marking as keywords 368
- test plan 589
- test plan attributes
 - Set attribute, adding members 132
 - Set attribute, removing members 132
- test plan editor
 - adding comments 122
 - predefined attributes 132
 - symbol definition statements 130
- test plan outlines
 - change levels 115
 - indent levels 115
- test plan queries
 - creating 136
 - overview 135
- test plan results
 - adding comments 115
 - generating pass/fail reports 476
- test plan templates
 - inserting 116
- test plans
 - acquiring locks 120
 - adding comments to results 115
 - adding data 121
 - adding data-driven test cases 195
 - assigning attributes and values 133
 - attributes and values 131
 - categorizing 125
 - changing colors 117
 - connecting sub-plans with master plans 119
 - copying sub-plans 119
 - creating 114
 - creating queries 136
 - creating sub-plans 119
 - creating, from results files 114
 - details, test cases 141
 - dividing into master plan and sub-plans 118
 - documenting manual tests 116
 - editor statements 122
 - example outline 112
 - generating completion reports 120
 - indent and change levels in outlines 115
 - inserting templates 116
 - large test plans 118
 - linking 123
 - linking manually to a test plan 124
 - linking scripts to using the Testplan Detail dialog box 124
 - linking test cases to using the Testplan Detail dialog box 124
 - linking to data-driven test cases 124
 - linking to scripts 117, 124
 - linking to test cases 117, 124
 - linking to test cases, example 125
 - locks 120
 - marking 125
 - marking tests 126
 - marking-command interactions 126
 - opening sub-plans 119
 - overview 111
 - predefined attributes 132
 - printing marked tests 126
 - queries 135
 - refreshing local sub-plan copies 119
 - releasing locks 120
 - running 445
 - sharing initialization files 120
 - stopping 550
 - structure 111
 - templates 112
 - user defined attributes 132
 - working with 114
- test results
 - interpreting 449
 - reporting 220
 - viewing 55, 450
- test scripts
 - creating, best practices 153
 - debugging 478, 479
- test-cases
 - working with data-driven 185
- testcase statements
 - specifying symbols as arguments 131
- TestCaseEnter method
 - overriding default recovery system 106

- TestCaseExit method
 - overriding default recovery system 106
- testcases
 - designing 173
 - overview 172
 - stopping during execution 447
 - types 173
- testdata statement
 - entering manually 123
 - entering with Testplan Details dialog box 122
- testing
 - application states 154
 - concurrency 265
 - configuration 266
 - databases 223
 - driving multiple machines 211
 - functional 266
 - peak load 266
 - strategies 264
 - volume 266
- testing .NET applications
 - Open Agent 267
- testing Apple Safari
 - information service, installing 309, 310, 342, 344
- testing applications
 - invalid data 188
 - Open Agent 203
- testing asynchronous in parallel 214
- testing controls
 - Web applications 332
- testing custom controls
 - Flex 254
- testing images
 - Web applications 333
- testing Java
 - configuring Silk Test Classic 287
 - prerequisites 287
- testing links
 - Web applications 333
- testing multiple applications
 - overview 223
 - window declarations 225
- testing multiple machines
 - overview 217
 - running tests serially on multiple targets 218
- testing serially
 - client and server 222
- testing text
 - Web applications 334
- testing web applications
 - web page objects, Open Agent 331
- testing Web applications
 - specifying browser 93
 - testing text 334
- Testplan
 - dialog box 114
- Testplan Completion Report
 - dialog box 121
- Testplan Detail
 - dialog box 140
- Testplan Detail - Testcase
 - dialog box 141
- Testplan Detail dialog box
 - defining symbols 130
 - linking scripts to test plans 124
 - linking test cases to test plans 124
- Testplan Details dialog box
 - entering testdata statement 122
 - linking descriptions to scripts and test cases 123
- testplan editor
 - # operator 122
- Testplan Editor
 - predefined attributes 132
 - statements 122
- testplan pass-fail report and chart 452
- testplan queries
 - overview 135
- TestPlanEnter method
 - overriding default recovery system 106
- TestPlanExit method
 - overriding default recovery system 106
- tests
 - marking 126
 - porting to another GUI 405
 - recording actions 151
 - running 443
 - running and interpreting results 443
 - slowing down 441
- text boxes
 - Return key 409
- text click recording
 - overview 439
- text fields
 - return key 409
- text recognition
 - overview 439
- textContents
 - xBrowser 353
- threads
 - concurrent programming 211
 - logging 475
 - specifying target machines 219
 - synchronizing with semaphores 212
- timestamps
 - wrong, cross-browser tests 355
- timing options
 - Agent Options dialog box 495
- tips and tricks
 - data-driven test cases 186
- TotalMemory parameter 589
- Traditional Chinese 590
- transcript
 - enabling 484
- Transcript dialog box 485
- transparent classes
 - setting, recording 152, 488, 505
- trapping the exception number 525
- troubleshooting
 - 4Test Editor does not display enough characters 549
 - Apache Flex 519
 - basic workflow issues, Open Agent 519
 - configuration test failures 267
 - custom error handling 523
 - Eclipse 300

- error messages 519
- exception handling 523
- general tips 554
- invalidated-handle error 356
- Java applications 291, 534
- Java SWT 300
- mobile 317
- Open Agent 519
- other problems 541
- projects 81, 551
- recognition 553
- Silverlight 280
- testing on multiple machines 534
- Web applications 555
- window not found 522
- writing an error-handling function 529
- XPath 168
- troubleshooting Unicode content
 - characters not displayed properly 429
 - compile errors 430
 - dialog boxes cannot display multiple languages 428
 - fonts look different 429
 - IME looks different 430
 - only English when clicking Language bar icon 430
 - only pipes are recorded 428
 - only pipes can be entered in files 428
 - pipes and squares 428
 - pipes and squares are displayed in Win32 AUT 429
 - pipes and squares in the Project tab 428
 - Save as dialog box when saving existing files 430
 - Unicode characters do not display 429
- TrueLog
 - limitations 457
 - options for the Classic Agent 514
 - options for the Open Agent 517
 - prerequisites 457
 - replacement characters for non-ASCII 457
 - setting options 458
 - wrong non-ASCII characters 457
- TrueLog Explorer
 - about 457
 - modifying your script to resolve Window Not Found Exception 460
 - overview 456
 - setting options 458
 - toggling at runtime using a script 459
 - viewing results 459
- TrueLog Options - Classic Agent
 - dialog box 514
- TrueLog Options - Open Agent
 - dialog box 517
- TrueLog Options dialog box
 - modifying your script to resolve exceptions 460
 - opening 458
- type
 - statements 412
- typographical errors
 - debugging 486

U

- unable to connect

- agent 521
- unexpected Click behavior
 - Internet Explorer 357
- unexpected values
 - global variables 486
- unicode content
 - configuring Microsoft Windows XP PC 426
 - using DB Tester 423
- Unicode content
 - installing language support 426
 - setting up IME 427
 - support 422
 - troubleshooting 428
 - troubleshooting display issues 428
 - troubleshooting file format issues 429
 - troubleshooting IME issues 430
- uninitialized variables
 - debugging 487
- unique data
 - specifying 122
- Unix display
 - Rumba 324
- Update Files
 - dialog box 157
- Update Manual Tests
 - dialog box 143
- upload app
 - Mac 301
- uploading
 - keyword libraries 376
 - libraries 376
- user data directories
 - Google Chrome 347
- user defined attributes
 - test plans 132
- user-defined methods
 - documentation examples 437
- using basic workflow
 - enabling extensions 86
- using file functions
 - adding information to the beginning of a file 550

V

- values
 - assigning to test plans 133
 - finding and replacing 191
 - test plans 131
- variable
 - definition 590
- variables
 - changing values, debugging 483
 - executing keyword-driven tests 374
 - same name 486
 - viewing, debugging 482
- verification logic
 - adding to scripts while recording 179
- verification options
 - Agent Options dialog box 496
- verification properties
 - defining 399
- verification statement 590

- verifications
 - adding to scripts 179
 - defining properties 399
 - fuzzy 182
 - overview 179
- verifying
 - Apache Flex scripts 235, 254
 - control no longer displayed 183
 - object properties 179
 - window no longer displayed 183
- verifying appearance
 - bitmaps 180
- verifying bitmaps
 - overview 180
- verifying state
 - objects 181
- View Module dialog box 485
- View Options dialog box 455
- view trace listing
 - enabling 484
- viewing
 - test results 55, 450
- viewing an individual summary 473
- viewing class methods
 - Library Browser 437
- viewing files
 - associated with projects 76
- viewing resources
 - included within projects 76
- viewing results
 - TrueLog Explorer 459
- viewing statistics
 - comparing bitmaps 465
- virus detectors
 - conflicts 545
- Visual 4Test
 - definition 590
- Visual Basic applications
 - standard names 90

W

- wDynamicMainWindow object
 - DefaultBaseState 104
- web applications
 - different browsers 329
 - objects, testing 331
 - recording test frames, Open Agent 329
 - replay, specific browser 330
- Web applications
 - characters not displayed properly 429
 - columns 332
 - configuring 251
 - controls 332
 - custom attributes 167
 - empty page 555
 - error with IE and Accessibility 555
 - Extension Settings dialog box 95
 - images 333
 - links 333
 - no HTML elements 555
 - Open Agent 326
 - tables 332
 - testing text 334
 - troubleshooting 555
 - xBrowser test objects 328
- Web classes
 - not displayed in Library Browser 438
- web pages
 - testing objects, Open Agent 331
- Web testing
 - setting agent options 93
- WebSync 21, 22
- Win32
 - pipes and squares are displayed in AUT 429
 - priorLabel 363
- window declarations
 - assigning agents 26
 - improving 386
 - overview 385
 - recording for main window 387
 - recording only pipes 428
 - recording without 176
 - testing multiple applications 225
- window is not active 521
- window is not enabled 522
- window is not exposed 522
- window not found
 - troubleshooting 522
- window not found exceptions
 - preventing 48
 - setting in agent options 48
 - setting manually 48
- window part 590
- window timeout
 - setting 48
 - setting in agent options 48
 - setting manually 48
- windows
 - declarations 385
 - verifying that no longer displayed 183
- Windows 10
 - limitations 363
- Windows API-based applications
 - attributes 360
 - overview 360
 - testing 360
- Windows Forms
 - attributes 267
 - invoking methods 268
 - locator attributes 267
 - overview 267
 - priorLabel 271
- Windows Forms applications
 - custom attributes 168
- Windows Presentation Foundation
 - controls 271
 - locator attributes 271
 - overview, Open Agent 271
- Windows Presentation Foundation (WPF)
 - exposing classes 273
 - invoking methods 274
- Windows XP
 - unicode content 426

- WinForms
 - priorLabel 271
 - WinForms applications
 - custom attributes 168
 - workflow
 - data-driven 184
 - workflow bars
 - disabling 188
 - enabling 188
 - works order number 21, 22
 - WPF
 - class reference 277
 - classes that derive from WPFItemsControl 273
 - controls 271
 - custom controls 273
 - exposing classes 273
 - invoking methods 274
 - locator attributes 271
 - overview, Open Agent 271
 - sample application 271
 - WPF applications
 - custom attributes 168
 - WPF locator attributes
 - identifying controls 271
 - wrong timestamps
 - logs, cross-browser tests 355
 - WStartup
 - handling login windows (Open Agent) 107
- X**
- xBrowser
 - Apple Safari 342
 - browser configuration settings 340
 - changing browser type for replay 330
 - class and style not in locators 356
 - classes 360
 - configuring locator generator 338
 - cross-browser scripts 354
 - current browser type, viewing 355
 - Default BaseState 335
 - Dialog not recognized 356
 - DomClick not working like Click 355
 - exposing functionality 356
 - FAQs 353
 - FieldInputField.DomClick not opening dialog 356
 - font type verification 353
 - Google Chrome 345
 - innerHTML 353
 - innerText 353
 - innerText not being used in locators 354
 - Internet Explorer misplaces rectangles 355
 - link.select focus issue 355
 - locator attributes 335
 - Microsoft Edge 351
 - mouse move recording 356
 - Mozilla Firefox 348
 - navigating to new pages 355
 - object recognition 334
 - overview 326
 - page synchronization 336
 - playback, comparing API and native 339
 - recording an incorrect locator 355
 - recording locators 355
 - setting recording options 339
 - synchronization options, setting 337
 - test objects 328
 - textContent 353
 - wrong timestamps, logs 355
 - xBrowser testing
 - Apple Safari, limitations 343
 - current browser type, viewing 355
 - Microsoft Edge, limitations 351
 - xPath
 - supported subset 159
 - XPath
 - basic concepts 158
 - definition 590
 - sample queries 161
 - troubleshooting 168
- Z**
- Zoom window
 - capturing in scan mode 462
 - zooming windows
 - Bitmap Tool 465