Understanding DevPartner®

DevPartner Studio Professional Edition
DevPartner for Visual C++ BoundsChecker Suite

Release 8.1

COMPUWARE®

Table of Contents

Chapter 1

Introducing DevPartner

Chapter 2

Error Detection

Chapter 3

Static Code Analysis

Chapter 4

Automatic Code Coverage Analysis

Chapter 5

Finding Memory Problems

Chapter 6

Automatic Performance Analysis

Chapter 7

In-Depth Performance Analysis

Chapter 8

System Comparison

Appendix A

About DevPartner Studio Enterprise Edition and TrackRecord

Index

Preface

◆ Who Should Read This Manual

◆ What This Manual Covers

◆ Conventions Used In This Manual

◆ For More Information

This manual describes how to get started using your Compuware® DevPartner® Studio software.

Who Should Read This Manual

This manual is intended for new DevPartner users, and for users of previous versions of DevPartner who want an overview of new functions and interface changes.

This manual contains information relevant to all DevPartner products, including the Professional and Enterprise Editions, and the Visual C++ BoundsChecker Suite.

Note: The DevPartner memory analysis, static code analysis, and Performance Expert features analyze managed code only, and are therefore not supported in the DevPartner for Visual C++ BoundsChecker Suite.

New users should read Chapter 1 to get an overview of DevPartner Studio concepts and subsequent chapters to learn how to deploy individual suite components during a software development cycle.

Users of previous versions of DevPartner should read the Preface to the *Installing DevPartner* manual to see how this version of DevPartner differs from previous versions.

This manual assumes that you are familiar with the Windows interface and with software development concepts.

What This Manual Covers

This manual contains the following chapters and appendixes:

◆ **Chapter 1**, *Introducing DevPartner* describes the concepts and components of DevPartner, discusses how DevPartner fits into the software development cycle.

◆ **Chapter 2**, *Error Detection*, explains how to use DevPartner to uncover errors in your managed and unmanged C++ code.

◆ **Chapter 3**, *Static Code Analysis,* explains how DevPartner helps you locate a variety of errors in Visual Basic code.

◆ **Chapter 4**, *Automatic Code Coverage Analysis*, describes how you can use DevPartner to track how much code exericising an application undergoes.

◆ **Chapter 5**, *Finding Memory Problems*, describes how to use DevPartner to diagnose application anomalies that can be caused by misuse of memory and objects.

◆ **Chapter 6**, *Automatic Performance Analysis*, explains how DevPartner helps you locate bottlenecks, and code in need of optimization.

◆ **Chapter 7**, *In-Depth Performance Analysis*, explains how DevPartner helps you analyze a variety of full system performance issues.

◆ **Chapter 8**, *System Comparison,* describes how you can identify differences between computer systems to assist with troubleshooting application development problems.

◆ **Appendix A**, *About DevPartner Studio Enterprise Edition and TrackRecord*, explains how to use DevPartner Studio with Compuware enterprise tools.

Conventions Used In This Manual

This book uses the following conventions to present information.

◆ Screen commands and menu names appear in **bold typeface**. For example:

Choose **Item Browser** from the **Tools** menu.

◆ Computer commands and file names appear in `monospace typeface`. For example:

The *Understanding DevPartner* manual (`Understanding DevPartner .pdf`) describes...

◆ Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in `italic monospace type`. For example:

Enter `http://`*`servername`*`/cgi-win/itemview.dll` in the **Destination** field...

For More Information

You can use the feature-level online help to learn more about the DevPartner Studio software's functions and procedures.

View the DevPartner InfoCenter page from the **Start>DevPartner** meun to learn more about DevPartner Studio components. Manuals in Adobe Acrobat (`.pdf`) format are included.

◆ The *Installing DevPartner* manual provides What's New information, a detailed list of system requirements, and installation instructions.

◆ The *DevPartner Studio Quick Reference* provides an at-a-glance summary of DevPartner features accompanied by quick-start advice.

Chapter 1
Introducing DevPartner
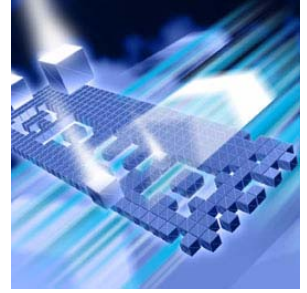
- ◆ Introducing DevPartner
- ◆ DevPartner Features
- ◆ DevPartner Support for Visual Studio
- ◆ DevPartner and Visual Studio
- ◆ DevPartner Methodology
- ◆ DevPartner Integration into the Enterprise

This chapter provides an introduction to version 8.1 of the DevPartner Studio Professional Edition, and DevPartner for Visual C++ Bounds-Checker Suite. It describes what you can use the DevPartner software to accomplish, and it explains what advantages such use provides.

Use this manual to understand the concepts underlying both the DevPartner products.

Introducing DevPartner

Programmers who want to take advantage of Internet connectivity without regard for hardware and device dependencies look to the Microsoft .NET Framework for a structure on which to build new applications and components. While this framework supplies these new structures, the problems of code migration, integration errors, coding errors, run-time performance bottlenecks and insufficient test coverage continue to plague software developers. DevPartner is designed to help not only with .NET migration, but with legacy integration, location of errors in application code, the tuning of run-time performance, and with more thorough test coverage.

What Is DevPartner?

DevPartner provides a variety of programmer productivity features, such as source code analysis, automated error detection, performance profiling, memory analysis, system performance analysis, and coverage analysis. These features let you analyze:

◆ Program logic problems
◆ Structured exception handling
◆ COM use count
◆ Non-standard coding practices
◆ Variable naming inconsistencies
◆ Code bottlenecks and inadequately tested code

Supports Multiple Version of Visual Studio

The DevPartner Studio software simultaneously supports application development within the Visual Studio 2003 and 2005 environments, and supports VC++ development in Visual Studio 6.0. This support assists developers as they migrate code from the older Microsoft environments to the latest .NET Frameworks.

DevPartner Features

This section summarizes the features of the DevPartner software. More detailed information about each feature can be found in subsequent chapters.

Error Detection

The DevPartner software provides automated error detection for managed and unmanaged C++ programs. DevPartner error detection is built on BoundsChecker™ technology, and is designed to locate the following hard-to-find errors in your Windows-based applications:

◆ Memory, resource, and COM interface leaks
◆ Invalid use of Windows API calls
◆ Invalid use of memory or pointers
◆ Memory overrun errors
◆ Un-initialized memory usage
◆ Use of dangling pointers
◆ Errors in .NET Finalizers

To give you a better understanding of the software you are developing, DevPartner error detection provides these additional types of application monitoring:

- Windows API Call/Return logging
- Graphical analysis of COM use counts
- Analysis of calls made from managed to native code

You can use DevPartner error detection on a wide variety of applications, including, but not restricted to the following:

- Traditional Windows applications
- In- and out-of-process COM servers
- ActiveX objects
- Windows services
- ISAPI filters

DevPartner has the following error detection features:

- Support for Visual Studio 2003 and 2005
  - Visual Studio IDE integration
  - FinalCheck support for Visual C++
  - Support for native and managed executables
  - .NET performance monitoring
  - .NET unmanaged exception monitoring
  - Finalizer analysis

- Cradle to grave monitoring of your application
  - DevPartner error detection monitors your application from the moment of creation until the final moments before the process is unloaded from memory
  - DevPartner error detection monitors all DLL loads and unloads, static constructors and destructors as well as the normal flow of your application
  - Cradle to grave monitoring provides complete visibility into your application
- COM use count graphical analysis
- Suppression and filtering system
- Modules and Files support to include or exclude portions of your application from analysis
- Settings enable you to:
  - Tune DevPartner error detection to collect only the information necessary to solve a particular problem
  - Make space vs. time trade-offs
  - Make performance vs. data collection trade offs
  - Reduce unwanted noise errors

See "Error Detection" on page 15 for more information about DevPartner error detection.

Static Code Analysis

The DevPartner software analyzes ASP.NET, Visual Basic .NET and Visual C# .NET source code to detect a variety of coding errors:

◆ Variable naming inconsistencies
◆ Violations of coding covenants
◆ Win32 API validation
◆ Common logic errors
◆ .NET portability issues
◆ Structured exception handling

Using an extensive and extensible rule set, DevPartner also can assist the porting of legacy Visual Basic code by identifying constructs that will not work in the .NET environment.

See "Static Code Analysis" on page 37 for more information about DevPartner static code reviews.

Performance Analysis

The DevPartner software can analyze your managed code applications, native C++ applications, ADO.NET components, or ASP.NET or Web applications for performance bottlenecks (see "Language/Technology Support" on page 7 for a complete list of supported technologies). It can pinpoint these bottlenecks to individual lines of source code, and provide method-level insight into the way your application uses third-party components, the operating system, and, most importantly, the .NET Framework.

DevPartner also supports performance profiling of legacy Visual Studio 6.0 components used by your .NET application. See "Automatic Performance Analysis" on page 131 for more information about analyzing an application's performance.

In-Depth Performance Analysis

DevPartner Studio contains many features designed to assist application development, including a performance analyzer that helps you locate bottlenecks in your code. Performance Expert takes performance profiling a step further for managed code Visual Studio applications by providing deeper analysis of the following hard-to-solve problems:

◆ CPU/thread usage

- File/disk I/O

- Network I/O

- Synchronization wait time

Performance Expert analyzes your application at run-time and locates the problem methods in your code. It then allows you to view details about individual lines in the method, or to examine parent-child calling relationships to help you determine the best way to fix the problem. When you have decided on an approach, Performance Expert enables you to jump directly to the problem lines in your source code, so you can quickly fix problems. See "In-Depth Performance Analysis" on page 153 for more information.

Coverage Analysis

DevPartner provides coverage analysis to assist developers and test engineers, ensuring that they are testing all of an application's code. DevPartner can collect coverage data for managed code applications, including Web and ASP.NET applications, as well as unmanaged (native) Visual C++ applications.

DevPartner Coverage Analysis gathers coverage data for applications, components, images, methods, functions, modules, and individual lines of code.

See "Automatic Code Coverage Analysis" on page 77 for more information about code coverage analysis.

Memory Analysis

DevPartner analyzes how memory is allocated by your managed Visual Studio application. When you run your application under memory analysis, DevPartner can show you the amount of memory consumed by an object or class, track the references that are holding an object in memory, and identify the lines of source code within a method responsible for allocating the memory.

More importantly, DevPartner presents memory data in context, enabling you to navigate chains of object references and calling sequences of the methods in your code, thereby providing both an in-depth understanding of how your program uses memory and the critical information you need to optimize memory use.

See "Finding Memory Problems" on page 101 for more information about memory analysis.

System Comparison

The DevPartner software includes a system comparison utility which runs outside of Visual Studio with a standalone user interface. The Compuware DevPartner System Comparison utility compares two computer systems, or compares the current state of your computer with a previous state, allowing you to determine why your application:

◆ Works on one computer but not on another
◆ Works differently on different computers
◆ No longer works on a computer on which it previously worked

The System Comparison utility takes snapshots of machine configurations, registry settings, system services, drivers, installed products, and so on, compares them, and reports the differences between snapshots.

The System Comparison utility includes a Software Development Kit (SDK) that allows software developers to gather additional information for comparison and to embed snapshot functionality in deployed applications.

See "System Comparison" on page 171 for more information about the System Comparison utility.

Interoperability

DevPartner provides interoperability between error detection and coverage analysis. You can gather information about errors and coverage at the same time. Performance analysis will always be done in a separate operation so as not to taint the performance data DevPartner collects.

Distributed Data Collection

DevPartner provides the facilities to collect memory, performance, system performance, and coverage data from machines remote to the user/console machine. This feature can be enabled for each remote/host machine that you would like to gather remote data from by:

◆ Installing your DevPartner software on the remote/host machine

◆ Obtaining and installing the optional DevPartner Server license, one license for each remote machine

Use these facilities if you have placed components of your distributed application on one or more remote/host machines and you would like to carry out memory, performance, system performance, or coverage analysis on the remote components. The data collected can be saved to a session file and its content analyzed on a DevPartner user/console

machine. DevPartner can even correlate data obtained from your user/ console and remote/host machines into a single sessions view.

See "Collecting Server-side Coverage Data" on page 85, "Collecting Coverage Data from Remote Systems" on page 86, "Collecting Server-side Performance Data" on page 139, and "Collecting Performance Data from Remote Systems" on page 140 for more information about remote data collection.

DevPartner Support for Visual Studio

This section describes the support DevPartner 8.1 provides for Visual Studio and the .NET Framework.

Web Services Analysis

DevPartner can perform runtime memory, performance, and coverage analysis on Web services. When Web services are present, DevPartner displays a Web Methods icon in the filter pane.

Language/Technology Support

DevPartner provides feature support for a variety of languages and technologies in various combinations, including:

◆ Visual Basic .NET
◆ ASP.NET
◆ Visual C# .NET
◆ Visual C++ .NET
◆ JScript .NET
◆ ADO.NET
◆ Web Forms
◆ Windows Forms
◆ XML Web services
◆ .NET Object Remoting
◆ Unmanaged C/C++ (Visual C/C++ 6)
◆ Native COM objects
◆ Win32 API calls
◆ ActiveX
◆ Mobile SDK
◆ ATL server
◆ COM callable wrappers
◆ NT services
◆ Out of Process COM servers

Managed vs. Unmanaged Code

DevPartner can analyze and profile both managed and unmanaged (native) C/C++ code, including mixed mode applications. For example, the DevPartner software's error detection, performance analysis, and coverage analysis capability allows the collection of data for a managed code application that includes native C/C++ code, provided the native code is in a separate file.

Configuration

DevPartner provides configuration options in Visual Studio. These options include control of suppressions and exclusions, lists of modules and files to target for analysis, and many other options.

Instrumentation Model

DevPartner inserts hooks into the code you write to trap and report on error conditions and to compute performance metrics and coverage statistics. The instrumentation of unmanaged code is identical to that of earlier versions of DevPartner.

The common language runtime in Visual Studio requires a slightly different instrumentation model. Here, DevPartner inserts hooks into the intermediate byte code stream.

DevPartner provides an Instrumentation Manager to specify the type of instrumentation DevPartner uses when you rebuild a native (unmanaged) Visual C/C++ project for data collection.

Visual Studio Team System Support

Visual Studio 2005 Team System is Microsoft's version control, defect tracking, and process management software for Visual Studio 2005 software development projects. DevPartner Studio supports Microsoft Visual Studio Team System if the Team System client software is installed and a Team Foundation Server connection is available.

DevPartner Studio supports submission of a **Work Item** of the type **Bug** to Visual Studio Team System. When you submit a **Bug**, DevPartner automatically populates the **Work Item** form with selected session data. In order to submit a **Bug** from DevPartner Studio, the active Team System project must support a **Work Item** of the type **Bug**. DevPartner Studio automatically adds data only to this type of **Work Item**.

You can submit a **Work Item** that includes DevPartner data from any of the following views in a DevPartner session file.

- A method list or method table in a Coverage, Memory, or Performance Analysis session file, or in a Performance Expert session file

- The code review **Problems** or **Naming** tabs

- A list of errors or leaks in any Error Detection tab, or list of instances in the Error Detection **Modules** or **.NET Performance** tabs

For more information about submitting data from DevPartner Studio to Team System, see the Visual Studio Team System topics in the DevPartner Studio online help. Consult the Microsoft Visual Studio 2005 Team System documentation for complete information on how to use Team System to support your development and project management activities.

DevPartner and Visual Studio

This section describes the DevPartner software's integration into Visual Studio, and explains its basic usage model.

IDE Integration

DevPartner integrates seamlessly into the Visual Studio environment. This integration makes it easy for you to use the capabilities of the product as you write and debug your .NET applications. You can perform these code analyses frequently as you develop an application without leaving the development environment.

Menus and Toolbars

DevPartner adds a menu and several toolbars to Visual Studio, and it adds menu commands to several Visual Studio menus, including context (right-click) menus. Menu commands and toolbars provide access to session controls, the rules for static code reviews, options dialogs, and instrumentation controls.

Using DevPartner

The general work flow for using DevPartner within Visual Studio consists of one or more of these general-purpose tasks:

- Open or create a Solution in Visual Studio
- Set Options for code analysis operations
- Enable the analysis you want to perform from the DevPartner menu or toolbar
- Run your application
- View the session results returned by DevPartner

DevPartner gives you wide flexibility in choosing what parts of your application to monitor, selecting what data to view, and creating filters to eliminate unwanted information.

DevPartner also gives you the option to perform many functions from the command line. This capability provides a way to use DevPartner functionality in automated batch processing operations, such as nightly-build smoke tests.

General User Interface Concepts

In addition to the menu and toolbar add-ins described earlier, DevPartner uses the Visual Studio dockable windows and panes to display the results of analysis sessions. It also uses the Solution Explorer to display information about .NET projects, such as the names of session files. DevPartner also adds pages to Visual Studio Options for configuring DevPartner code analysis operations.
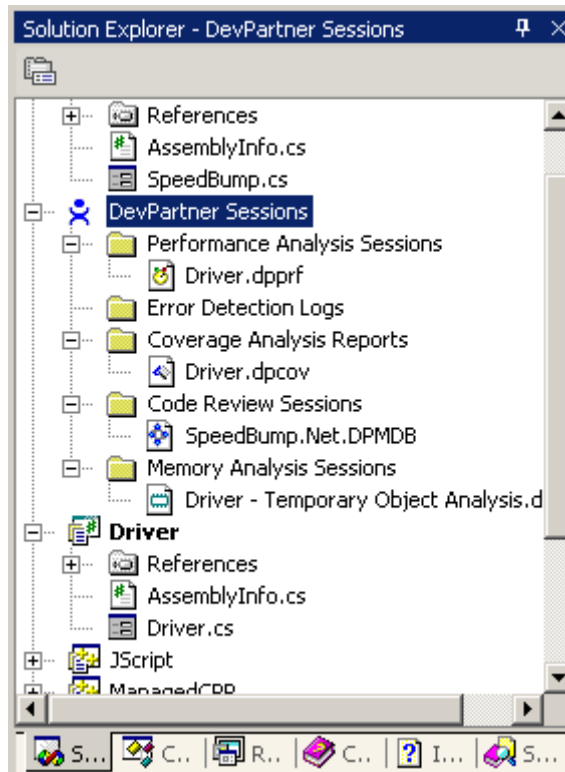
Figure 1-1. Solution Explorer with DevPartner Nodes

Integrated Online Help

DevPartner provides extensive online help about each of its code analysis features. This help should be the first place you turn for how-to and reference information.

Provided in the same format as the rest of Visual Studio help, the DevPartner online help appears in the Visual Studio help collection as a separate book that in turn contains several volumes, including books for error detection, performance analysis, coverage analysis, memory analysis, system performance analysis and static code analysis.

DevPartner Methodology

This section describes when to use the DevPartner software, and provides some typical usage scenarios.

While software development life cycles may differ somewhat among development organizations, the life cycle described in this section portrays one way that DevPartner can be used over the course of a project. DevPartner adapts to virtually any development life cycle model, whether a company uses a rolling releases life cycle model, or a classic waterfall model.

Software Development Life Cycle Model

Software development projects consist of several phases. Sometimes phases are discrete, while others may overlap, such as when the product wish-list for an application revision accumulates while the initial product finishes its external test cycle. In any given organization, the number of life cycle phases may vary.

The following figure depicts a project that breaks down into five development life cycle phases: *Define, Plan, Develop, Test,* and *Deploy.*

Tip: Organizations may define the actions between phases as project milestones.

Definition and Planning

Within each phase, teams perform a variety of tasks and often repeat tasks to refine the desired end result. For example, in the first two phases, *Define* and *Plan*, teams may use a variety of tools from word processing, spreadsheet, and project management tools to more sophisticated

requirements planning tools. DevPartner Enterprise Edition offers additional, project-tracking capabilities that provide solutions for the first two phases of the development life cycle. *Develop*, *Test*, and *Deploy*, which are often the more time-consuming and precarious phases, can also benefit from the DevPartner project management capabilities.

Developing the Application

The *Develop* phase is where the construction of the software application begins and progresses. Some activities might parallel others during this phase, such as:

◆ Systems analysts set up workspaces and programming tool environments.
◆ Software developers construct code.
◆ Quality assurance engineers develop test and automation scripts.
◆ Release engineers initiate and manage daily builds.
◆ Technical communicators write user manuals and online help.
◆ Usability specialists verify the ease-of use of the user interface.
◆ Project management supervises the overall development.
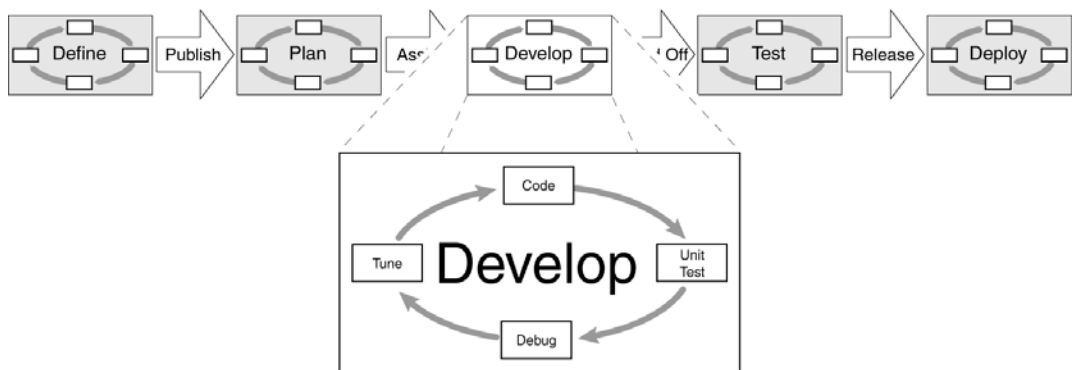
Figure 1-2. Iterative Tasks of Development

DevPartner is an integral part of the Develop phase. It helps developers find and resolve software defects, as well as fine-tune and test the application under development. Information generated by DevPartner features can be shared among development team members to foster communication. DevPartner provides unique benefits for each functional group within the development organization whose needs may differ significantly throughout the software implementation process.

Testing the Application Internally and Externally

All development organizations use internal load testing and scenario-based testing to verify the operation of features in an application under development, and this internal testing continues until the end of the development life cycle. DevPartner provides many advantages during this phase of the development cycle.
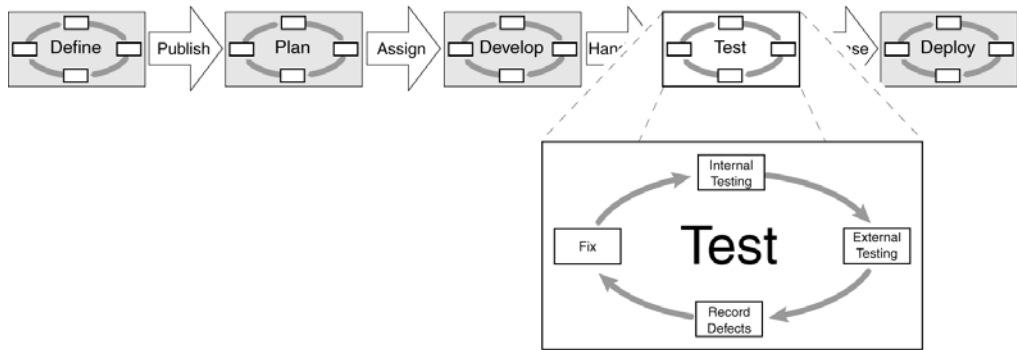
Figure 1-3. Test Phase

Using an active analysis technology, DevPartner error detection and performance analysis features can align with Compuware *QA*Center test tools, such as *QA*Run and *QA*Load, to provide supplemental advantages to streamline the application testing process.

In addition, field tests outside the immediate development organization are crucial to the success of software projects. They permit testing under conditions that are not typical within the development group. External tests expose your application to platforms, network setups, and other conditions that would be impossible to anticipate and duplicate in-house. For example, the DevPartner software's code coverage analysis feature helps guarantee adequate testing coverage.

Deploying the Application to End Users

Using DevPartner, a development team can successfully build and release its application with a high degree of confidence in the final product release. Inevitably, however, internal or external customers may find problems that even the most sophisticated technologies fail to uncover. Since such problems can adversely affect the end-user experience, your development team needs to address them when they arise. DevPartner Enterprise Edition helps you manage this process with its full repertoire of defect detection, verification, and resolution capabilities.

As a team grows in size or an application grows in complexity, the additional defect tracking and integration technologies of the DevPartner Enterprise Edition can further enhance an organization's productivity during and after deployment.
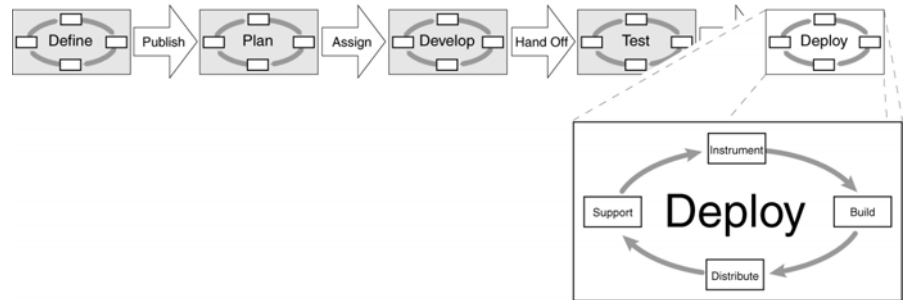
Figure 1-4. Deployment Phase

DevPartner Integration into the Enterprise

The DevPartner Enterprise Edition, which includes the TrackRecord and Reconcile applications, provides integration between DevPartner and Compuware TrackRecord. This integration is provided through ActiveLink technology. See "About DevPartner Studio Enterprise Edition and TrackRecord" on page 179 for more information about using DevPartner in an enterprise environment.

Chapter 2
Error Detection

This chapter describes the basic features of DevPartner Error Detection. DevPartner Error Detection automates the crucial process of error-detection and analysis, identifies elusive bugs that are beyond the reach of traditional debugging and testing techniques, and adds little or no time to the development process.

For information that goes beyond the basics, refer to *Advanced Error Detection Techniques*, provided as a PDF with the DevPartner software installation.

Check Early, Check Often—The Best Error Detection Philosophy

To increase software quality, developers must thoroughly test their code early in the development process. Bugs must be caught and resolved as they are introduced to avoid surprises during integration, quality assurance, beta testing, and production. Briefly stated, "check early, check often."

The Benefits of Using DevPartner Error Detection

DevPartner Error Detection is the most comprehensive, automated debugging solution available for C and C++ development. If you develop Windows applications, you will benefit from these DevPartner Error Detection features:

◆ Comprehensive error detection
◆ Flexible debugging environment
◆ Integration with Microsoft Visual C++ 6.0 and Visual Studio
◆ Advanced error analysis
◆ Open error-detection architecture

Comprehensive Error Detection

DevPartner Error Detection can analyze Windows applications with both ActiveCheck™ and FinalCheck™ technologies.

ActiveCheck

DevPartner Error Detection uses ActiveCheck technology in all error detection sessions. ActiveCheck detects errors in your program without requiring you to recompile or relink.

ActiveCheck can do the following:

◆ Report API validation errors at run-time
◆ Report memory and resource leaks when your program terminates
◆ Isolate errors to the line where the memory or resource was allocated or the error was generated
◆ Identify potential deadlocks

Note: DevPartner Error Detection always enables ActiveCheck technology even when FinalCheck is also selected.

When you run your program under DevPartner Error Detection, it automatically analyzes your program as it runs. DevPartner Error Detection monitors your program's API calls, memory allocations and deallocations, windows messages, and other significant events, then uses this data to detect errors and to provide a complete trace of your program's execution. You can even check programs that do not have source code available.

Because ActiveCheck requires no compilation or relinking overhead, you can use it daily. Use ActiveCheck throughout the software development cycle to find API validation errors, deadlocks, resource leaks, and COM interface leaks.

Table 2-1 and Table 2-2 list errors detected by ActiveCheck.

Table 2-1. API and COM and Memory errors detected by ActiveCheck

API and COM Errors	Memory Errors
• COM interface method failure	• Dynamic memory overrun
• Invalid argument	• Freed handle is already unlocked
• Invalid COM interface method argument	• Handle is already unlocked
• Parameter range error	• Memory allocation conflict
• Questionable use of thread	• Pointer references unlocked memory block
• Windows function failed	• Stack memory overrun
• Windows function not implemented	• Static memory overrun

Table 2-2. Deadlock-related, .NET, and Pointer and Leak errors detected by ActiveCheck

Deadlock-related Errors	.NET Errors	Pointer and Leak Errors
• Deadlock	• Finalizer errors	• Interface leak
• Potential deadlock	• GC.Suppress finalize not called	• Memory leak
• Thread deadlocked	• Dispose attributes errors	• Resource leak
• Critical section errors	• Unhandled native exception passed to managed code	
• Semaphore errors		
• Mutex errors		
• Event errors		
• Handle errors		
• Resource usage and naming errors		
• Suspicious or questionable resource usage		
• Windows event errors		

FinalCheck

FinalCheck is a patented technology that DevPartner Error Detection can use to instrument Visual C or Visual C++ applications. FinalCheck inserts diagnostic logic into your code when you compile it. With FinalCheck, DevPartner Error Detection can pinpoint errors to the exact statement where they occurred.

Use FinalCheck for key project milestones and for detecting errors that are difficult to find.

FinalCheck is a superset of ActiveCheck that finds all the errors ActiveCheck finds, plus those listed in Table 2-3.

Table 2-3. Additional Errors Detected by FinalCheck

Memory Errors	Pointer and Leak Errors
• Reading overflows buffer • Reading uninitialized memory • Writing overflows buffer	• Array index out of range • Assigning pointer out of range • Expression uses dangling pointer • Expression uses unrelated pointers • Function pointer is not a function • Memory leaked due to `free` • Leak due to leak • Memory leaked due to reassignment • Memory leaked leaving scope • Returning pointer to local variable • Leak due to unwind • Leak due to module unload • Leak due to thread ending

An Example Comparing ActiveCheck and FinalCheck

If you allocate a block of memory using `new` or `malloc` and store the pointer in a local variable, DevPartner Error Detection will record that information. If you later re-assign another value into the local variable without first either deallocating the memory block or assigning the pointer to another variable, you have just created a leak in your application.

◆ **Using ActiveCheck:** DevPartner Error Detection reports that the block allocated by `malloc` or `new` was leaked and points to the line where the memory was allocated. The error is reported when your application exits.

◆ **Using FinalCheck:** DevPartner Error Detection reports the location where the block was allocated and highlights the line where you assigned the new value into the last remaining variable referencing the block. The error is reported when it occurs.

Flexible Debugging Environment

DevPartner Error Detection provides a flexible debugging environment which can be run:

◆ As an integrated part of Microsoft Visual Studio
◆ As an independent application
◆ From a DOS command line

When you use DevPartner Error Detection as part of the Visual Studio environment, all of its features can be accessed from within the IDE. You can configure DevPartner Error Detection settings, check your program, and review detected errors.

When you use DevPartner Error Detection as an independent application, it runs completely outside of the Microsoft IDE.

When you use DevPartner Error Detection from a DOS command line, you can set up automated testing scripts. See "Running DevPartner Error Detection from the Command Line" on page 35 for more information.

Integration with the Visual Studio Debugger

DevPartner Error Detection automatically integrates with the Visual Studio debugger.

The **Program Error Detected** dialog box includes a **Debug** button. When you click **Debug**, DevPartner Error Detection drops into the IDE debugger, at the line of code that generated the error.

To review logged errors in the **Details** pane of the DevPartner Error Detection window, right-click on an event and select **Edit Source**. This opens the source file at the line of code that generated the error.

Advanced Error Analysis and Event Logging

Windows is an event-driven environment in which much of your program is executed in response to Windows messages and other events. DevPartner Error Detection intercepts control when events occur and logs them. You can use these logs to see a complete history of events that led to a problem.

DevPartner Error Detection logs the following events:

◆ Windows messages and hooks.
  These events show how your program reacted to Windows messages.
◆ API calls and API returns along with argument information.
  These events define the order in which procedures are executed in
  your program.
◆ Output debug string messages from the program you are checking.
◆ Error messages, including all information DevPartner Error Detection
  recorded in the event log.

Open Error Detection Architecture

You can extend DevPartner Error Detection in two ways:

◆ By describing your user-written allocators to the memory tracking
  system by adding lines to UserAllocators.dat. Refer to *Working with
  User-Written Allocators* in *Advanced Error Detection Techniques*.

◆ By selecting **Generate NLB files dynamically**, DevPartner Error
  Detection will automatically learn about:

  ◇ COM interfaces that are defined in modules which contain type
    library information

  ◇ .NET classes in modules that contain metadata information

  Once added, you can log COM and .NET method calls and returns.
  You can also obtain detailed information about COM interface leaks.

DevPartner Error Detection Main Window

The DevPartner Error Detection main window is divided into three
sections, called panes:

◆ Results pane
◆ Details pane
◆ Source pane

See Figure 2-1 on page 21.

Results Pane

The upper left section of the window is the **Results** pane. The **Results**
pane uses a series of tabs for navigating through the various types of
information.

The **Summary** tab (first tab on the left of the **Results** pane) provides an overview of all errors and events detected in your session. Double-click on a specific event to navigate to its corresponding entry in the **Memory Leaks**, **Other Leaks**, or **Errors** tab.

Results pane ——————   Details pane ——————
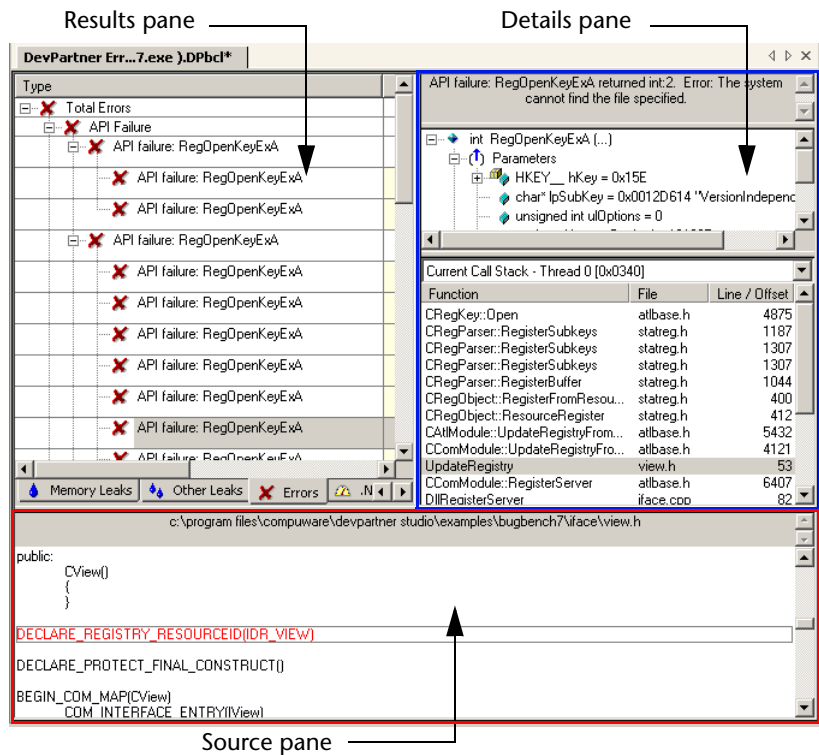
Figure 2-1. The DevPartner Error Detection Main Window

The rollup tabs include **Memory Leaks**, **Other Leaks**, **Errors**, **.NET Performance**, **Modules**, and **Transcript**.

◆ To sort the data on these tabs, click a column header (such as **Type**, **Quantity**, or **Deallocator** in the **Other Leaks** tab).

◆ For additional information about an event on a tab, right-click the event and choose **Explain.**

◆ To see an event in the context of other events in your application, right-click the event and choose **Locate in Transcript** from the shortcut menu. The **Transcript** tab provides a chronological list of all events that occurred within your application.

Details Pane

The upper right section of the window is the **Details** pane. The **Details** pane consists of one or more sections, depending on the currently selected event. The top section describes the error or event in detail. The lower section(s) can display call stacks, P/Invoke use-count graphs, COM use-counts, and so on.

When you select an event, the **Details** pane displays the current call stack. If more than one call stack is accessible, the **Details** pane provides a drop-down list; use it to select a different call stack to view.

Source Pane

The bottom section of the window is called the **Source** pane. It displays the source file associated with the currently selected call stack frame displayed in the **Details** pane. The source code changes when you select a different call stack in the **Details** pane.

Settings Dialog Box

The DevPartner Error Detection settings enable you to:

◆ Select only the types of data collection needed for a particular problem

◆ Enable or disable portions of each major type of data collection

◆ Control what portions of your program are analyzed

◆ Use the default DevPartner Error Detection settings to find the most common errors with the minimum impact on performance

The **Settings** dialog box has a tree control that shows the major settings categories. When you select a category, the dialog box displays the detailed settings for the category.

Note: In Visual Studio, the term **Options** is used instead of **Settings**.

The same tree control and settings dialog boxes are used in the DevPartner Error Detection standalone application and in the Visual Studio IDE.

All groups of settings follow the same basic structure. You can enable or disable major types of data collection by selecting the top-level check-box in the dialog box.

There are other settings under each top-level check box that further define how DevPartner Error Detection will analyze your application. Change the settings to customize your error detection process.

For example, you can make trade-offs between detecting a broad or narrow range of errors:

◆ Broad range — Many data types, many related settings selected
  ◇ Detects more errors
  ◇ Has potential for more false positives
  ◇ Reduces performance (due to larger number of errors detected)
  ◇ Creates larger log files

◆ Limited range — Few data types, few related settings selected
  ◇ Provides a narrow focus on a particular function
  ◇ Detects fewer errors
  ◇ Can miss relevant errors
  ◇ Has a greater chance of seeing only those errors pertaining to the problem at hand
  ◇ Provides faster performance
  ◇ Creates smaller log files

An Example: Call Validation

To activate **Call Validation**, select **Enable call validation**. This makes all the **Call Validation** controls active. (By default, **Call Validation** is turned off.)
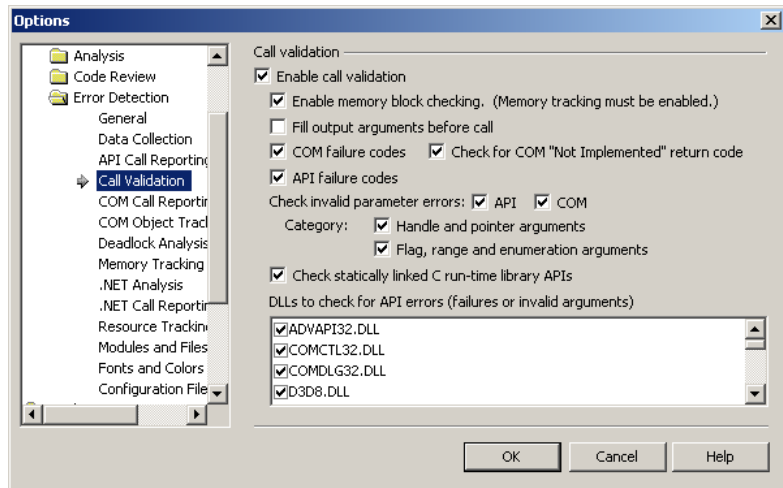
Figure 2-2. The **Settings** or **Options** Dialog Box Showing the Controls Under **Call Validation**.

When you enable Call Validation, DevPartner Error Detection validates over 5,000 Windows API calls. DevPartner Error Detection will check for a large number of events, including (but not limited to) the following:

◆ Handle and pointer errors
◆ Flags
◆ Range checks
◆ API and method failures
◆ Invalid structure sizes
◆ Memory access failures

If you find flag checking or range checking generates unwanted errors that do not apply to the problem you are solving, clear the **Flag**, **range and enumeration arguments** check-box. Call Validation will continue checking return values and, more importantly, handles and pointers passed to or from Windows calls.

Enable Memory Block Checking

By selecting **Enable memory block checking**, Call Validation will perform a more detailed analysis of all calls to the C run-time library and a number of other calls. By default, this setting is inactive. Selecting **Enable memory block checking** will decrease overall performance but may prove useful when diagnosing hard-to-find errors.

Using Other Settings Categories

The online help provides detailed information about each settings category and also describes some of the trade-offs associated with specific tools within a category.

Program Error Detected Dialog Box

DevPartner Error Detection displays the **Program Error Detected** dialog box (see Figure 2-3 on page 25) when it detects an error in your application.

The top of the **Program Error Detected** dialog box describes the error detected. Below this will be one or more tabs. Each tab is associated with a call stack corresponding to a location within your application. Review the reported error and the source information to help you locate the source of the problem and correct it.
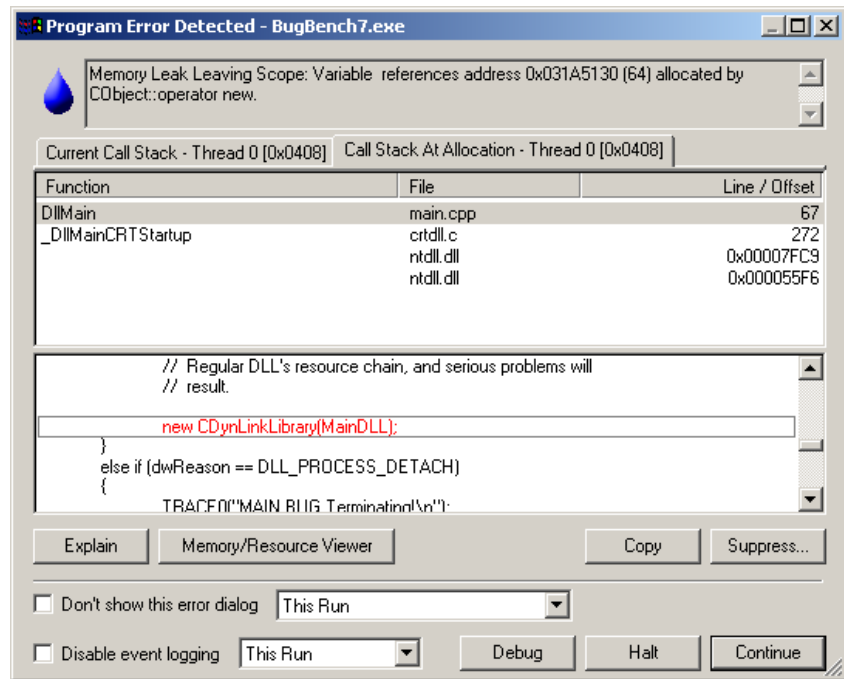
Figure 2-3. The **Program Error Detected** Dialog Box

Buttons on the Program Error Detected Dialog Box

Explain, **Memory and Resource Viewer**, **Debug**, **Copy** and **Suppress** buttons appear on the **Program Error Detected** dialog box. If you have installed DevPartner Studio Enterprise Edition with TrackRecord integration, you also have a **Submit** button available.

Explain

Click **Explain** to obtain detailed explanations of each error along with sample code and a list of solutions to correct the problem.

Memory and Resource Viewer

Click **Memory and Resource Viewer** to view a detailed accounting of memory and resources that have not been freed. For more information, see "Memory and Resource Viewer Dialog Box" on page 27.

Submit

Submit is only available if TrackRecord is part of your DevPartner installation. Click **Submit** to open TrackRecord to either a new defect or new task page.

Copy

Click **Copy** to transfer the contents of all windows and tabs (except the **Source** pane) to the clipboard. You can then paste this information into other applications.

Suppress

Click **Suppress** to open a dialog box that enables you to suppress the current error. For more specific instructions, click **Help** in the suppression dialog box.

Debug

Debug appears at the bottom of the dialog box when you are working from within Visual Studio or Visual C++.

Click **Debug** to drop into the Visual Studio debugger. You can then examine variables or modify the source.

Halt

Click **Halt** to stop the application.

Continue

Click **Continue** to close the dialog box and continue executing the application.

Memory and Resource Viewer Dialog Box

The **Memory and Resource Viewer** dialog box allows you to analyze memory and resource allocations that have not been freed.

For example, most analysis tools can determine that memory or resources have been leaked only at the end of an application. This information tells you little about usage during the middle of a program's execution. The DevPartner Error Detection **Memory and Resource Viewer** can provide a snapshot taken at any point in a program's execution. You can also "mark" the currently allocated memory blocks or resources, limiting the view of blocks allocated after a program's initialization or over the course of a transaction.

These capabilities can be especially useful in situations like these:

◆ 24/7 server applications may never end during regular use
◆ An application may hang from resource exhaustion
◆ An application may consume large amounts of memory that is automatically cleaned up at program termination
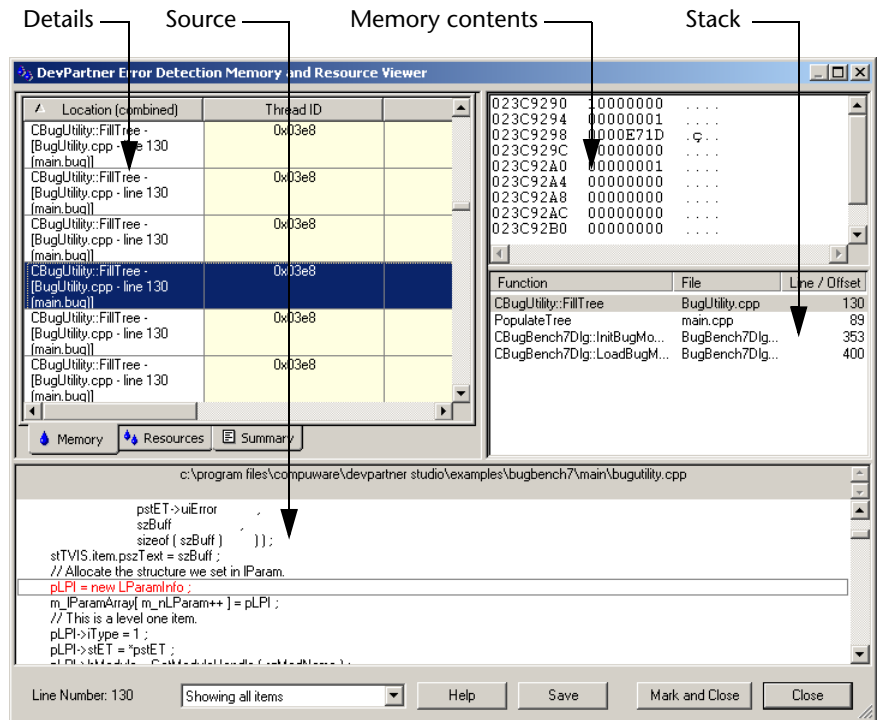
Figure 2-4. The **Memory and Resource Viewer** Dialog Box.

The Memory and Resource Viewer User Interface

To access the **Memory and Resource Viewer** dialog box, click **Memory/Resource Viewer** in the **Program Error Detected** dialog box.

The **Memory and Resource Viewer** dialog box is made up of four panes:

◆ **Memory contents pane**
Displays the content of memory blocks in a variety of formats. Not available for resources.

◆ **Details pane**
Includes separate Memory, Resources, and Summary tabs. Displays details about each memory and resource allocation.

◆ **Stack pane**
Displays a memory dump and callstack information for entries in the Memory tab; displays a description and callstack information for entries in the Resources tab.

◆ **Source pane**
Displays the source code corresponding to a callstack entry (when it is available).

Saving Memory and Resource Viewer Contents

Click **Save** to record the current contents of the **Memory and Resource Viewer** dialog box as a text file that you can review later.

Mark and Close

Click **Mark and Close** to set a reference point for recording memory and resource data. This enables you to compare memory and resource allocations before and after the event where you marked the reference point.

Suppression and Filtering Dialog Boxes

Suppression and filtering allow you to reduce the data collected or displayed. The intent of either method is to limit the data to a manageable subset for analysis.

Suppressing Errors

By suppressing errors, you instruct DevPartner Error Detection to skip over any future occurrences of those errors. Suppressed errors are not recorded in the log and they are not displayed in the **Program Error Detected** dialog box. Use one of the following actions to suppress an error:

◆ Right-click on an error in one of the panes and select **Suppress**
◆ Select an error and click the **Suppress** button on the toolbar

Filtering Errors

Filtering hides events already recorded in a .DPBCL log file. DevPartner Error Detection finds these errors but either hides them from view in the **Results** pane or displays them with the appearance you specified under **Fonts and Colors**. Use one of the following actions to select errors that you want to filter:

◆ Right-click on an error in one of the panes and select **Filter**
◆ Select an error and click the **Filter** button on the toolbar

If you cancel a filtering instruction, the filtered errors appear in the Results pane.
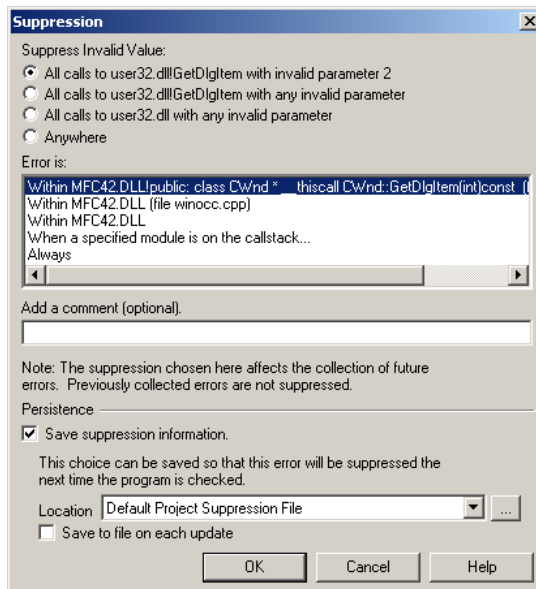
Figure 2-5. The **Suppression** Dialog Box (The **Filtering** Dialog Box Uses the Same Design).

Suppression and Filtering User Interface

Toggle filtering and suppression on and off using the toolbar icons.

The DevPartner Error Detection **Suppression** and **Filter** dialog boxes provide numerous controls over the suppression and filtering processes.

For example, you can suppress call validation errors from FindResourceA in Kernel32 or for all calls in Kernel32. After you make this selection, you can apply it to a variety of different selection criteria within your application. DevPartner Error Detection defaults to the least restrictive option (see Figure 2-5).

You can make the following additional choices for suppressions and filters:

◆ Enter a comment to describe why a given suppression or filter was created.

◆ Choose to apply the suppression or filter to the current run or future runs.

◆ Create suppression or filter files to store the suppression or filter instructions.

Creating and Saving Suppression and Filter Files

You can create multiple suppression files, and in doing so create additional suppression libraries for the various DLLs that make up a large application. You can easily reuse or share suppressions among members of a development team.

See the online documentation for detailed information on each field in the **Suppression** and **Filter** dialog boxes.

DevPartner Error Detection Integration with Visual Studio

DevPartner Error Detection is tightly integrated into Visual Studio menus, toolbars, the **Solution Explorer**, the debugger and the build system.

◆ Choose **DevPartner -> Options** to open the **Options** dialog box, then select **Error Detection** under **DevPartner** in the tree view to see the DevPartner Error Detection options (see Figure 2-6).
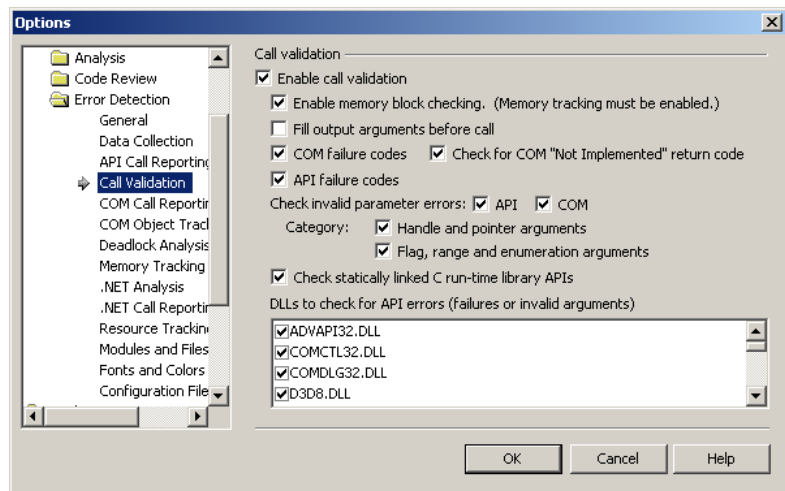
Figure 2-6. The **Options** Dialog Box.

◆ Choose **DevPartner -> Start with Error Detection** to start debugging your project with **Error Detection**. Commands on this menu control FinalCheck instrumentation and other DevPartner functions (see Figure 2-7).
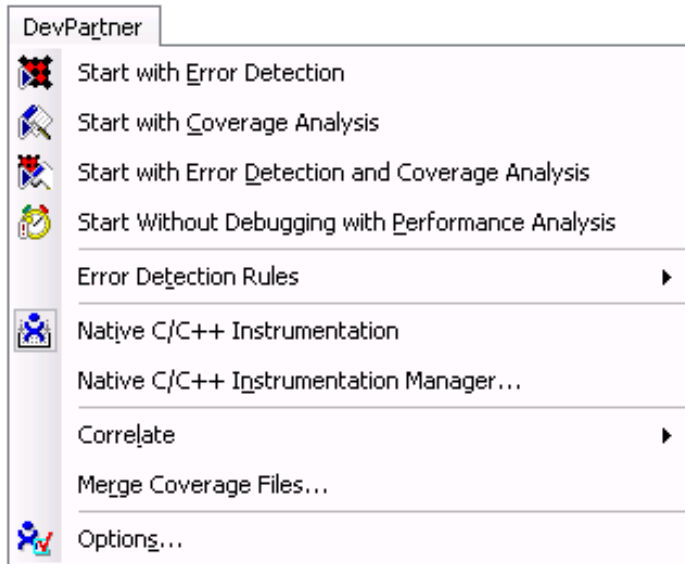
Figure 2-7. The DevPartner Menu in Visual Studio.

◆ You can use the error detection toolbar (see Figure 2-8) to select the instrumentation desired, change any settings, and start running with Error Detection.
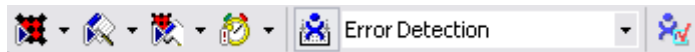
Figure 2-8. The DevPartner Toolbar.

◆ DevPartner Error Detection will automatically register its sessions with the **Solution Explorer** (see Figure 2-9).
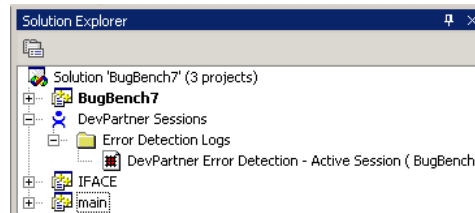
Figure 2-9. DevPartner Error Detection Sessions are Automatically Registered with the Solution Explorer.

Using DevPartner Error Detection with Visual Studio 2003/2005

Follow these steps to use DevPartner Error Detection with Visual Studio 2003/2005:

1 Create or open an existing solution.

2 Build your application.

To analyze your program with ActiveCheck, create an executable with these attributes:

◇ A Debug build, preferably with optimizations disabled

◇ With debug symbols

◇ With Visual C++ basic run-time checking disabled, do one of the following:
   − Remove /RTC from the CL command line
   − Set Basic runtime checks in the C/C++ Code Generation settings to Default

To analyze your native C/C++ program with FinalCheck, re-compile your application. You can instrument your program by doing one of the following:

◇ Enable FinalCheck instrumentation using one of the following procedures:
   − Click the instrumentation button on the DevPartner toolbar
   − Choose **DevPartner -> Native C/C++ Instrumentation**

◇ Build your application with Visual Studio

If you have solutions with multiple projects, you may want to use the DevPartner Error Detection **Instrumentation Manager** to select which projects should be instrumented. In this case, choose **DevPartner -> Native C/C++ Instrumentation Manager**.

3 Choose the DevPartner Error Detection options you want to use.

◇ Choose **DevPartner -> Options** to access the **Options** dialog box, then select **Error Detection** under **DevPartner** in the tree view. Review the DevPartner Error Detection options.

For example, to validate Windows API calls made by your application, select **Call Validation**. Under **Call Validation**, select **Enable call validation**. For most applications, the default settings should produce acceptable results.

4 Select **DevPartner -> Start with Error Detection** to start debugging. As your program executes, DevPartner Error Detection will display any errors encountered.

5 Review the reported errors, then click either **Debug** or **Continue**.

6 Review errors in the **Results** pane of the DevPartner Error Detection window after your program terminates. To correct an error, right-click an event and choose **Edit Source**.

7 Correct your errors, recompile your application, and continue testing.

Integration with Visual Studio 6

DevPartner Error Detection also integrates into the Visual Studio 6 menus, toolbars, debugger, and build system. See Figure 2-10.
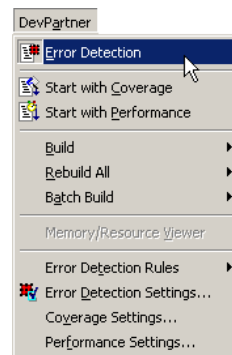
Figure 2-10. The DevPartner Error Detection menu in Visual Studio 6.

Follow these steps to run DevPartner Error Detection while debugging your application with Visual Studio:

1 Create or open an existing project.

2 Build your application.

   To analyze your program with ActiveCheck, create an executable with debug symbols and, preferably, with optimizations disabled.

   To analyze your program with FinalCheck, instrument your program:

   ◇ Select the debug configuration

   ◇ Select either **DevPartner -> Build -> Error Detection** or **DevPartner -> Rebuild -> Error Detection**

      This action will re-compile your application. The **Build** tab of the Output window will display messages stating that DevPartner Error Detection is instrumenting your application.

3 Choose the DevPartner Error Detection settings to use for the analysis:

◇ Select **DevPartner -> Error Detection Settings**, then make selections in the **Settings** dialog box

Example: To validate Windows API calls made by your application:

– Select **Call Validation** from the tree view in the settings dialog box

– Select **Enable call validation**

For most applications, the default settings produce acceptable results.

4 Press **F5** to start debugging.
As your program executes, DevPartner Error Detection will display errors in the **Program Error Detected** dialog box.

5 Review the reported errors, then click **Debug** or **Continue**.

6 Review errors in the **Results** pane of the DevPartner Error Detection window after your program terminates. To correct an error, right-click the event in the **Details** pane and choose **Edit Source**.

7 Correct your errors, recompile your application, and continue testing.

Running DevPartner Error Detection from the Command Line

You can run DevPartner Error Detection from a DOS command line, using `bc.exe` or `bc.com`.

Note: For legacy support of the 7.x versions, DevPartner Error Detection allows you to continue using `bc7.com` in your script files.

◆ `bc.exe` starts the UI for DevPartner Error Detection standalone.

◆ `bc.com` is a small console program that spawns `bc.exe` and waits for it to complete.

The difference between `bc.exe` and `bc.com` is important for batch scripts. Invoking `bc.exe` directly will start DevPartner Error Detection and continue on to the next command without waiting for `bc.exe` to complete. If the next step in the script is to check for a result, it won't be available.

Note: If you type only `bc`, the OS will choose `bc.com` instead of `bc.exe`.

For more information, refer to Using Error Detection from the Command Line in *DevPartner Advanced Error Detection Techniques*.

Running FinalCheck from the Command Line
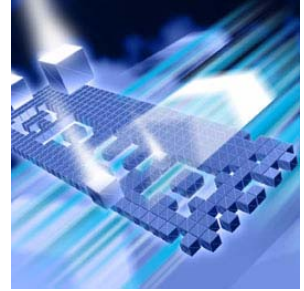
You can also run FinalCheck from the command line. For more information, refer to the following topics in the *Checking a Program with FinalCheck* section of the online help.

◆ Running FinalCheck from the Command Line
◆ NMCL Options
◆ NMLINK Options

Chapter 3
Static Code Analysis

◆ DevPartner Code Review

◆ DevPartner Code Review User Interface

◆ Naming Analysis Functionality

◆ Call Graph Analysis Functionality

◆ DevPartner Code Review Rule Manager

◆ Creating New Rules Using Regular Expressions

This chapter describes the DevPartner code review feature. DevPartner helps developers write compliant Visual Basic .NET and Visual C# code within the Visual Studio Integrated Development Environment (IDE). DevPartner identifies programming and naming violations in the .NET Framework, analyzes method call structures, and tracks overall code complexity.

Note: The DevPartner code review feature analyzes managed code only, and is therefore not supported in the DevPartner for Visual C++ BoundsChecker Suite.

DevPartner Code Review

The DevPartner code review feature delivers the following functionality:

◆ Static code analysis and review

    DevPartner performs a comprehensive static code analysis of your source code and displays results in the **DevPartner Code Review** window that is integrated in the IDE.

◆ Automated command-line batch processing

DevPartner lets you execute a command line batch review of your solution in conjunction with a nightly build or as an alternative to reviewing large applications.

◆ Rules management and customization

The Rule Manager lets you configure rules, triggers, rule sets, and name sets used by the Hungarian naming analyzer.

DevPartner Code Review User Interface

The DevPartner code review feature integrates in the Visual Studio IDE. The **DevPartner Code Review** window is the main user interface component (Figure 3-1 on page 39) and is comprised of these tabbed results panes:

Table 3-1. DevPartner Code Review Window — Results Tabs

Tab	Displays
Summary	Results and count summaries (see "Summary Pane" on page 39)
Problems	Programming problems and additional information (see "Problems Pane" on page 46)
Naming	Naming violations (see "Naming Pane" on page 47) (Optionally includes Naming Details pane for the Naming Guidelines naming analyzer only (see "Naming Details Pane" on page 50)
Metrics	Evidence of code complexity (see "Metrics Pane" on page 51)
Call Graph	Graphical method or property call structures in current solution (see "Call Graph Pane" on page 53)
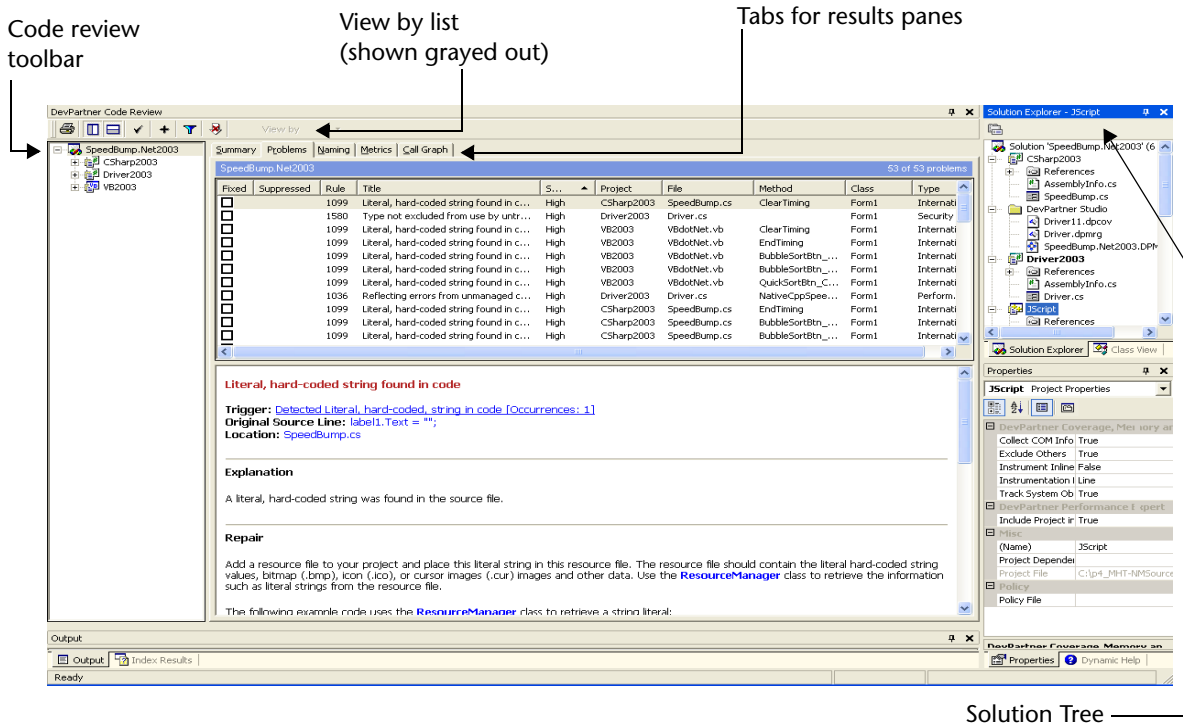
Figure 3-1. DevPartner Code Review Window

Summary Pane

The **Summary** pane consolidates summarized results data in a single location. The **Summary** pane is divided into the following tables:

- Summary of Problems (Figure 3-2 on page 40)
- Summary of Naming Guidelines (Figure 3-3 on page 41)
- Summary of Call Graph Data (Figure 3-4 on page 42)
- Summary of Counts (Figure 3-5 on page 42)
- Review Settings (Figure 3-6 on page 43)
- Project List (Figure 3-7 on page 45)

Some contents on the **Summary** pane are dynamic. As items on the **Problems** or **Naming** results panes are marked as *Fixed*, the corresponding sections on the **Summary** pane will dynamically reflect the update. If the review was performed with some exceptions, such as bypassing compile errors or a build requirement, the **Summary** pane will reflect that message in the header section.

Note: Consult the DevPartner Code Review online help for information on how compile errors and build requirements affect a code review.

Summary of Problems lists the rule-based categories that were covered in the review.

Type / Names	Problems		Severity			
	Total	Fixed	High	Medium	Low	Warning
COM Interop	1	0	0	0	0	1
Database	0	0	0	0	0	0
Date	0	0	0	0	0	0
Design Time Properties	0	0	0	0	0	0
Error/Exception Handling	21	0	0	1	20	0
Garbage Collection	0	0	0	0	0	0
Internationalization	12	0	12	0	0	0
Language	0	0	0	0	0	0
Logic	2	0	0	0	0	2
Maintainability	13	0	0	2	3	8
Performance	1	0	1	0	0	0
Portability	0	0	0	0	0	0
Project & Solution Properties	0	0	0	0	0	0
Reliability	0	0	0	0	0	0
Security	3	0	3	0	0	0
Standards	0	0	0	0	0	0
System	0	0	0	0	0	0
Usability	0	0	0	0	0	0
User-Defined Rule	0	0	0	0	0	0
Versioning	0	0	0	0	0	0
Windows API	0	0	0	0	0	0
Totals	**53**	**0**	**16**	**3**	**23**	**11**

* Summaries include all rule violations. Your filter settings do not apply.

Figure 3-2. **Summary of Problems** Table on **Summary** Pane

Summary of Problems provides the following information:

Table 3-2. Contents of Summary of Problems

Categories	Description
Types	List of reviewed rule-based categories
Total problems	Count of problems found
Total fixed problems	Total number of problems fixed on the **Problems** pane; dynamic update
Severity	Total number of problems categorized by severity: • **High**, **Medium**, **Low** set by priority • **Warning** used to call attention to a unique development issue

Note: Details about problems uncovered during a code review are provided on the **Problems** pane (Figure 3-8 on page 46).

Summary of Naming Guidelines lists the categories that you originally selected on the Naming Guidelines property page (Figure 3-15 on page 55) to be included in the review.

Note: This table is only displayed on the **Summary** pane if the **Naming Guidelines** naming analysis selection on the General Options property page (Figure 3-14 on page 54) was chosen prior to the review. It gives a summary of the naming identifiers selected on the Naming Guidelines property page. This table is not available for the Hungarian naming analyzer.

Summary of Naming Guidelines		
Naming Identifiers	**Total Reviewed**	**Total Violations**
Variables	100	83
Parameters	18	8
Namespaces	2	2
Classes	1	0
Interfaces	0	0
Methods	24	10
Enumerations	2	0
Structures	0	0
Delegates	0	0
Events	0	0
Totals	**147**	**103**

Figure 3-3. **Summary of Naming Guidelines** Table on **Summary** Pane

Summary of Naming Guidelines provides the following information:

Table 3-3. Contents of Summary of Naming Guidelines

Category	Description
Naming Identifiers	Lists the naming identifiers previously selected from the **Include naming analysis for** list on the Naming Guidelines property page (Figure 3-15 on page 55)
Total Reviewed	Gives a count of the items that were reviewed, based on each naming identifier category
Total Violations	Subtotals the violations that were found, based on each naming identifier category

Note: More information about naming guidelines-related violations appears on the Naming Details pane just below the Naming pane (Figure 3-10 on page 49). The Naming Details pane is not available for the Hungarian naming analyzer.

Summary of Call Graph Data lists summarized information about call graph analysis that was captured during the review.

Note: This table is only displayed on the **Summary** pane if the **Collect Call Graph Data** selection on the General Options property page (Figure 3-14 on page 54) was selected prior to the review.

Summary of Call Graph Data	
Summary Type	**Count**
Total Methods Graphed	24
Total Methods Uncalled	0

Figure 3-4. **Summary of Call Graph Data** Table on **Summary** Pane

Summary of Call Graph Data provides the following information:

Table 3-4. Contents of Summary of Call Graph Data

Categories	Description
Total methods graphed	Count of total methods/properties analyzed during the code review
Total methods uncalled	Count of uncalled methods/properties (see "Call Graph References" on page 63)

Note: Call graph data is graphically displayed on the **Call Graph** pane following a review (see "Call Graph Analysis Functionality" on page 62) .

Summary of Counts includes individual statistics and counts.

Summary of Counts	
Summary Type	**Count**
Review Time (in minutes)	1,212
Total Lines (including blank lines)	2,183
Code Only Lines	1,162
Comment Only Lines	270
Code with Comments	0
Rule Comparisons Made	468,267
Total Lines Checked	2,183

Figure 3-5. **Summary of Counts** Table on **Summary** Pane

The following table lists the statistics under **Summary of Counts**:

Table 3-5. Contents of Summary of Counts

Categories	Description
Review Time	Total review time in minutes
Total Lines	Total number of all lines in the solution including code, comment, and blank lines (comment lines are counted in methods only)
Code Only Lines	Total number of code lines only without trailing comments
Comment Only Lines	Total number of comment lines only without any code
Code with Comments	Total number of comment lines with code
Rule Comparisons Made	Total number of rule comparisons processed during the review that pertain to the current rule set
Total Lines Checked	Total number of lines that were reviewed, including code lines with trailing comments and blank lines

Review Settings

Review Settings lists configuration and review-related data. This information is useful for record keeping and troubleshooting.

Figure 3-6. **Review Settings** Table on **Summary** Pane

The **Review Settings** table includes the following information:

Table 3-6. Contents of Review Settings

Review Categories	Description
Solution	File name of solution that was reviewed
Solution Path	Path location of the solution that was reviewed
Session File	File name of session created following code review; either default name (`solution-name.dpmdb`) or renamed by user
Batch Command Execution File	Name of batch file created with code review
Reviewer	Name of user who initiated code review
Review Date	Date and time stamp of the review
DevPartner Code Review Version	Current version of DevPartner code review feature
Rules DSN	Rules database file name
Preferences DSN	Preferences database file name
Rule Set Used	Rule set applied to review
Count of Rules in Rule Set	Number of rules affecting review
Metrics Analysis	True if option was selected on the General Options property page (Figure 3-14 on page 54) False if not
Naming Analysis	Indicates what was selected at **Naming analysis to use** on the General Options property page (Naming Guidelines, Hungarian, or none)
Dictionary Name[1]	Indicates the dictionary used in the review
Identifiers Examined[1]	Indicates the .NET Framework identifiers that will be included in the review
Local Variables Examined[1]	True if local variables were examined False if not
Case used for locals[1]	User selection (Pascal or Camel case)
Company Name[1]	User entry
1. Applies to the Naming Guidelines naming analyzer only and selections previously made on the General Options and Naming Guidelines property pages	

Table 3-6. Contents of Review Settings (Continued)

Review Categories	Description
Technology Name[1]	User entry
Call Graph Analysis	True if Collect call graph data option was selected on the General Options property page False if not
Ignore compile errors	True if option was selected on the General Options property page False if not
Exclude rules that require a build	True if option was selected on the General Options property page False if not
Always generate a batch file	True if option was selected on the General Options property page False if not
1. Applies to the Naming Guidelines naming analyzer only and selections previously made on the General Options and Naming Guidelines property pages	

Project List **Project List** provides information for each project in the solution.

Project List			
Project Name	Compile Errors	Reviewed	Project Path
Driver2003	False	True	C:\p4_MHT-NMSource1666_MHT101515D01 \DPS\DP_Mainline\Analysis\Examples\SpeedBump.Net\Driver\Driver2003.csproj
CSharp2003	False	True	C:\p4_MHT-NMSource1666_MHT101515D01 \DPS\DP_Mainline\Analysis\Examples\SpeedBump.Net\CSharp\CSharp2003.csproj
VB2003	False	True	C:\p4_MHT-NMSource1666_MHT101515D01 \DPS\DP_Mainline\Analysis\Examples\SpeedBump.Net\VB\VB2003.vbproj

Figure 3-7. **Project List** Table on **Summary** Pane

The following table lists information under **Project List**:

Table 3-7. Contents of Project List

Category	What it covers
Project Name	Name of each project in the solution that was reviewed
Compile Errors	True if compile errors were detected in that project which could prevent the review from proceeding; false, if not
Reviewed	True if the review was executed on that project False if not
Project Path	Path location of that project

Problems Pane

By default, the **Problems** pane appears first in the **DevPartner Code Review** window following a code review. The **Problems** pane displays programming violations found in the current solution. This results view is divided into two adjoining sections — Problems pane (upper panel) and Description pane (lower panel).

Problems pane →
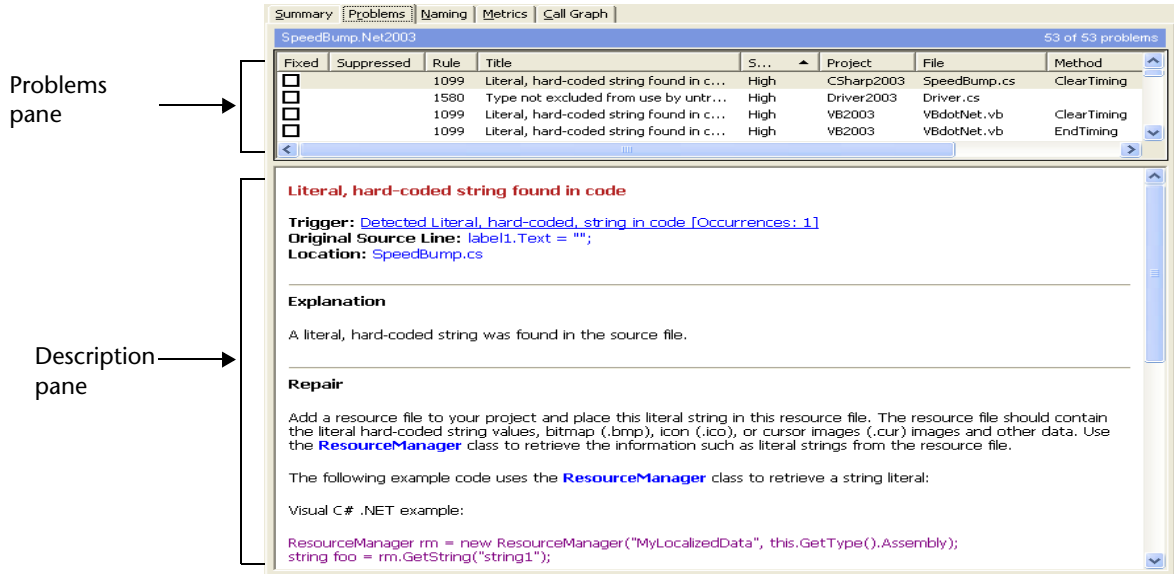
Description pane →

Figure 3-8. **Problems** Pane Including Problems Pane and Description Pane

Problems Pane (Upper Panel)

The Problems pane shows the rule violations that were detected. It appears in the upper panel.

Tip: Entries made in the Rule Manager appear in the Problems list following a DevPartner code review.

The following table contains the information provided on the **Problems** pane:

Table 3-8. Contents of Problems pane

Column	Description
Fixed	Status of the rule violation ☑ Checked for fixed, unchecked for not fixed
Suppressed	Status of the rule suppression *Suppressed*, or blank for not suppressed
Rule	Number assigned to that rule violation
Title	Title of the rule
Severity	Severity level (High, Medium, Low, Warning)

Table 3-8. Contents of Problems pane (Continued)

Column	Description
Project	Project where the violation exists
File	File where the violation exists
Method	Method where the violation exists
Class	Class of the rule that was fired
Type	Rule type

Description Pane (Lower Panel)

You select a rule violation on the Problems pane (upper panel) and its details appear on the Description pane (lower panel). The contents originate from rules stored in the DevPartner rules database (system-supplied and user-configured).

Tip: Each rule violation can include additional hyperlinks for Trigger, Original Source Line, and Location.

The following table lists the details provided on the Description pane:

Table 3-9. Contents of Description Pane

Heading	Description
Rule title (shown in red)	Title of the rule
Trigger	Name of the trigger; appears as a hyperlink that lets you go to the original source line
Original Source Line	Line of code that caused the rule to fire
Location	Identifies the origin of the rule violation
Explanation	Describes the rule violation
Repair	Recommends a remedy for the problem
Notes	Additional comments; optionally includes external links to Microsoft MSDN knowledge base articles

Naming Pane

The **Naming** pane lists naming violations that DevPartner finds during a code review. Its appearance varies depending on which type of naming analysis you selected on the General Options property page (Figure 3-14 on page 54) prior to the review. Regardless of the naming analysis picked, the upper panel of the **Naming** pane always lists the results.

Note: The **Naming** pane will display results from one or the other, but not from both naming analyzers. If **None** was selected, the **Naming** pane will be empty following a code review.

The next illustration shows how the **Naming** pane appears when the Hungarian naming analyzer was previously selected on the General Options property page (Figure 3-14 on page 54).
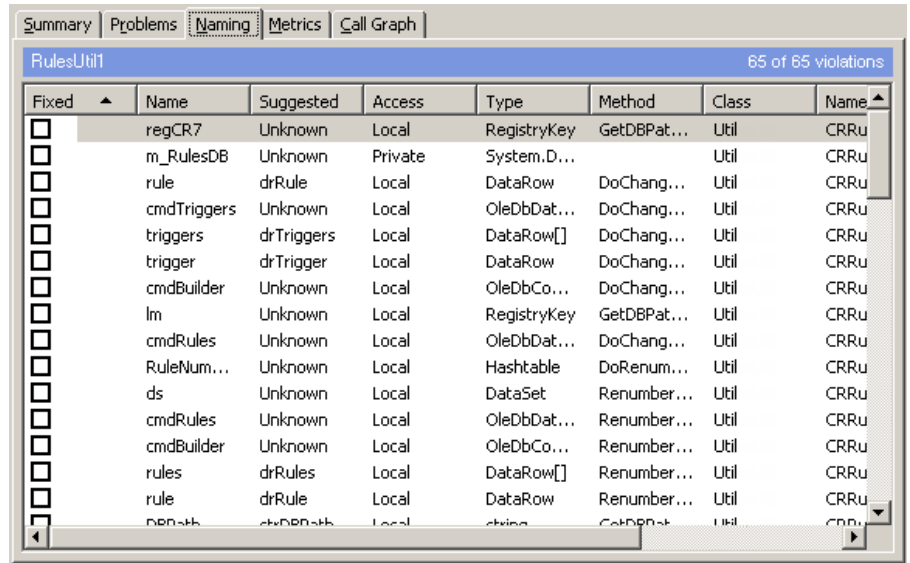
Figure 3-9. **Naming** Pane for Hungarian Naming Analysis

The next illustration shows a two-panel representation of the naming results when the Naming Guidelines naming analyzer was previously selected for the review. Notice the **Naming** pane in the upper panel and the Naming Details pane (Figure 3-11 on page 50) in the lower panel. Notice, also, the additional **View by** list (Figure 3-20 on page 58), located to the right of the toolbar.

Naming pane →

Naming Details pane →
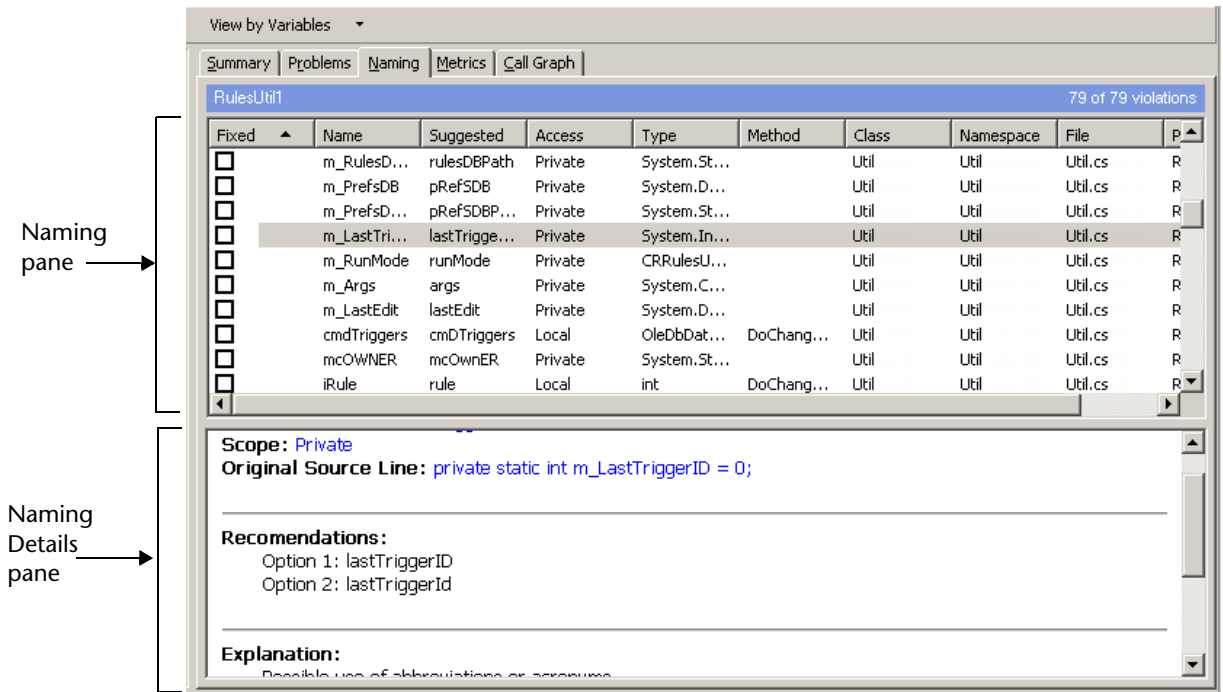
Figure 3-10. Two-Panel **Naming** Pane for Naming Guidelines Naming Analysis

The following table lists the information provided on the **Naming** pane (upper panel), regardless of which naming analyzer was selected:

Table 3-10. Contents of Naming Pane

Column	Description
Fixed	Status of the naming violation ☑ Checked for fixed, unchecked for not fixed
Name	User-defined name for the data type
Suggested	Suggested name Suggested names vary, depending on which naming analyzer was selected (see "Naming Analysis Functionality" on page 58) • If DevPartner cannot suggest a name based on Hungarian, *Unknown* will appear in this column. • If DevPartner cannot suggest a name based on Naming Guidelines, a series of asterisks will appear in this column. An explanation will also appear on the Naming Details pane (Figure 3-11 on page 50).
Access	Category of access within the current solution

Table 3-10. Contents of Naming Pane (Continued)

Column	Description
Type	Type of identifier
Method	Method where the data type is declared
Class	Class where the data type is declared
Namespace	Namespace where the data type is declared
File	File where the data type is declared
Project	Project where the data type is declared

Naming Details Pane

If you selected Naming Guidelines, plus made additional choices on the Naming Guidelines property page (Figure 3-15 on page 55), a Naming Details pane will also appear in the lower panel of the **Naming** pane.

Note: The Naming Details pane is only available for the Naming Guidelines naming analyzer, not Hungarian. See "Naming Analysis Functionality" on page 58 for more information about each naming analyzer.
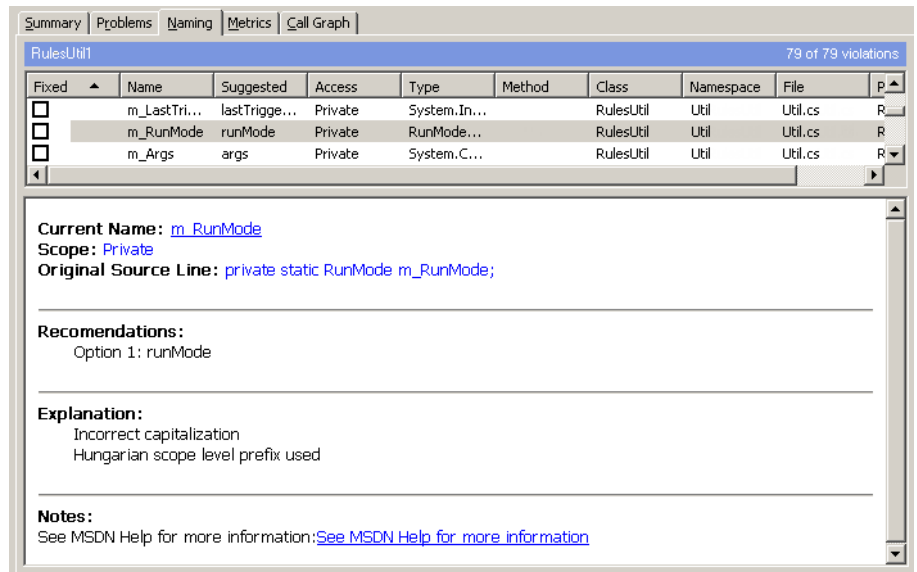
Figure 3-11. Naming Details Pane (for Naming Guidelines Naming Analysis Only)

Similar to the Problems Description pane (Figure 3-8 on page 46), the Naming Details pane provides additional details about the naming violation selected in the upper panel of the **Naming** pane.

When you click one of the .NET Framework naming violations listed on the **Naming** pane, the following information will appear on the Naming Details pane:

Table 3-11. Contents of Naming Details Pane

Item	Description
Current name	Corresponds to the item selected in the upper panel
Scope	Indicates the scope of the identifier
Original Source Line	Displays the source line that pertains to the selected naming violation in the upper panel
Recommendations	Suggests one or more suitable names, based on the Naming Guidelines naming analyzer (see "Naming Guidelines Naming Analyzer" on page 58)
Explanation	Provides an explanation for why this violation was flagged as a problem **Note**: If DevPartner cannot suggest a better name, an explanation will appear in this pane. DevPartner will also show a series of asterisks in the **Suggested** column of the upper panel of the **Naming** pane.
Notes	Optionally includes a hyperlink to the Naming Guidelines knowledge base in the .NET Framework General Reference

Metrics Pane

The **Metrics** pane (Figure 3-12 on page 52) displays code complexity results (Complexity, Bad Fix Probability, and Understanding Level), based on McCabe Metrics. Consult the DevPartner Code Review online help for more information about these metrics.

Figure 3-12. **Metrics** Pane

The **Metrics** pane only displays data if the **Collect metrics** check box on the General Options property page (Figure 3-14 on page 54) was selected prior to the review.

The following table lists the information provided on the **Metrics** pane:

Table 3-12. Contents of Metrics Pane

Heading	Description
Method	Method name where the code complexity issue originated
File	File name where the issue originated
Project	Project where the issue originated
Complexity	Indicates the degree of complexity regarding a particular component; this metric is related to McCabe Cyclomatic Complexity
Bad Fix %	Indicates the likelihood that a new bug will occur in the code when trying to fix a known bug
Understanding	Indicates how straightforward the code logic is to decipher and maintain
Lines of Code	Total lines of code within the selected component; breakdown of individual line counts appear on the **Summary** pane (see "Summary of Counts" on page 42)

Call Graph Pane

DevPartner integrates a **Call Graph** pane in the multi-tabbed **DevPartner Code Review** window.
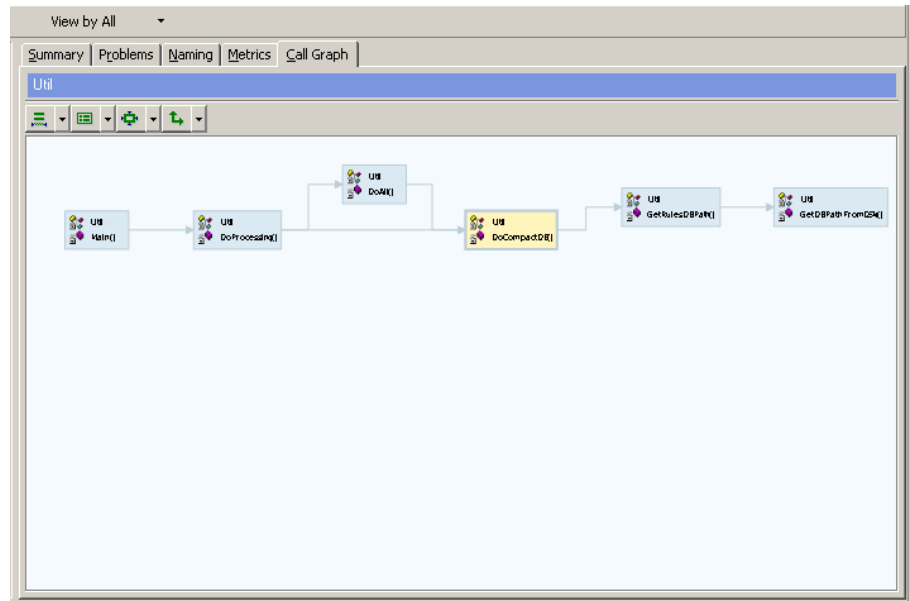
Figure 3-13. **Call Graph** Pane Showing Example of Call Graph Representation

Note: See "Call Graph Analysis Functionality" on page 62 for more information on this functionality. Consult the DevPartner Code Review online help to use this feature.

Code Review Options

DevPartner provides two property pages to modify code review options:

◆ General
◆ Naming Guidelines Options

Note: Consult the DevPartner Code Review online help for instructions on how to use these property pages.

General Options

The General Options property page contains code review settings that you can modify prior to your next code review. You can access this property page by selecting:

DevPartner > **Options** > **Code Review**
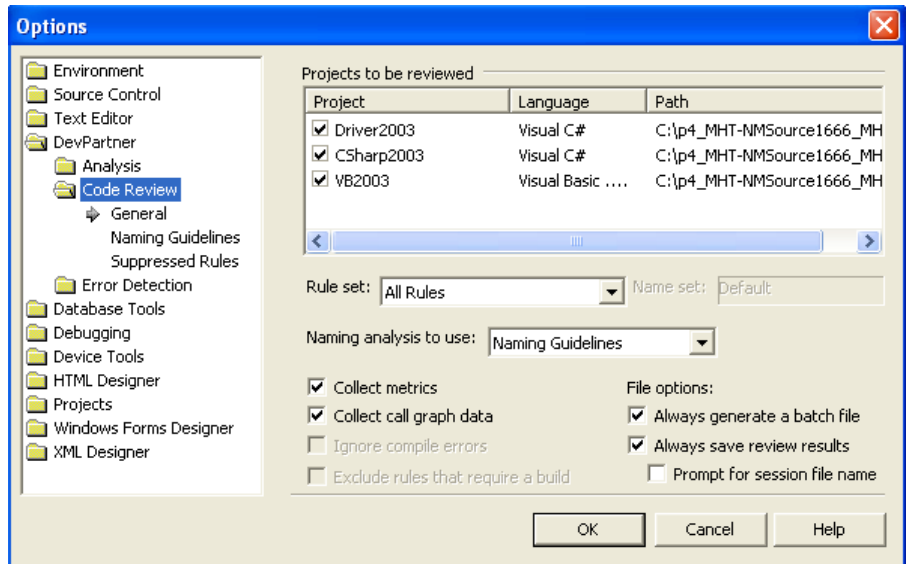
Figure 3-14. General Options Property Page for DevPartner Code Review

Naming Guidelines Options

The Naming Guidelines property page includes choices that ensure a more precise review. You can access this property page by selecting:

DevPartner > **Options** > **Code Review**

Note: Selections on this page are only available if you previously picked the **Naming Guidelines** naming analyzer from the **Naming analysis to use** list on the General Options property page (Figure 3-14 on page 54). Consult the DevPartner Code Review online help to use this property page.
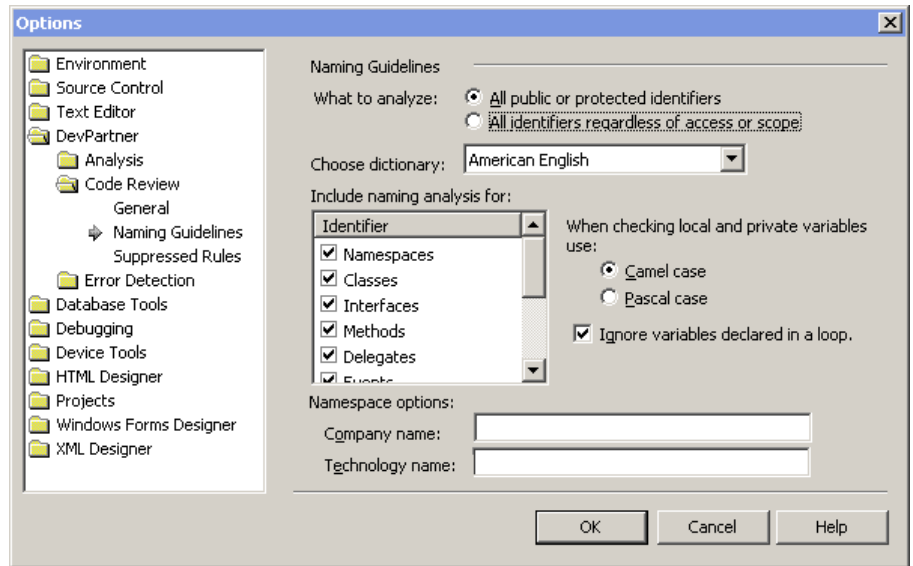
Figure 3-15. Naming Guidelines Property Page

DevPartner Code Review Toolbar Components

DevPartner provides three toolbars:

◆ Code review toolbar
◆ IDE toolbar
◆ Call graph toolbar

Code Review Toolbar

The **DevPartner Code Review** window includes a code review toolbar.
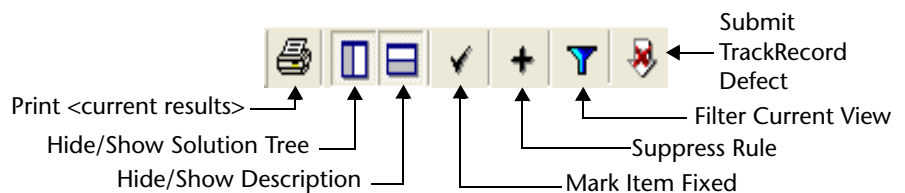
Figure 3-16. DevPartner Code Review Toolbar

IDE Toolbar

Optional toolbar buttons related to the DevPartner code review feature are available in the Visual Studio IDE. To access, right-click in the upper-right corner of the IDE and choose **DevPartner**.

Create and modify rules used during code reviews

Perform a review of the solution code
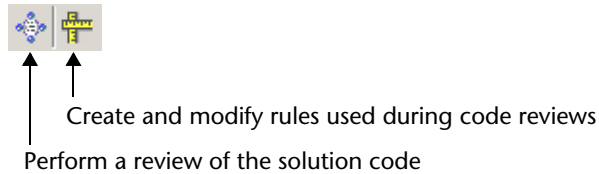
Figure 3-17. Additional Toolbar Buttons for Code Review

Call Graph Toolbar

A dedicated call graph toolbar is available when the **Call Graph** pane has focus. It is located just below the caption bar.

Layout

Scaling

Node style
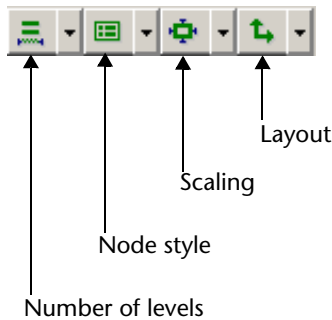
Number of levels

Figure 3-18. Call Graph Toolbar (Only Available with **Call Graph** Pane)

The toolbar provides the following configuration options:

Table 3-13. Call Graph Configuration Options

Button	Tooltip	Description
	Number of levels	Configures how many node levels will be displayed
	Node style	Configures the style of each node in the display area

Table 3-13. Call Graph Configuration Options

Button	Tooltip	Description
	Scaling	Configures how the entire call graph will fit in the display area
	Layout	Configures whether the layout of the call graph will be shown horizontally or vertically

Note: A context menu also provides the same configuration options. To access, click on the background area of the call graph (not on a node). See "Call Graph Configuration Options" on page 65 for a description of the configuration options. Consult the DevPartner Code Review online help for additional information.

View By Lists

A **View by** list appears to the right of the standard code review toolbar. The contents and functionality of this list will vary depending on whether the **Naming** or **Call Graph** pane has focus.

Filtered View for Naming Guidelines — If the **Naming** pane has focus, the **View by** list will only be visible if you previously selected the Naming Guidelines naming analyzer from the General Options property page (Figure 3-14 on page 54). You can filter the viewing results by choosing one of the .NET Framework naming identifiers from the **View by** list.

Note: Consult the DevPartner Code Review online help on how to make selections from the **Include naming analysis for** list on the Naming Guidelines property page (Figure 3-15 on page 55).

View by Variables
- Variables
- Namespaces
- Enumerations
- Interfaces
- Structures
- Classes
- Delegates
- Events
- Methods
- Parameters
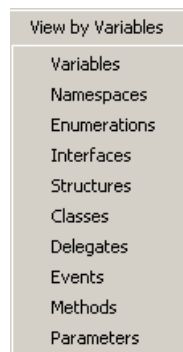
Figure 3-19. **View by** List for **Naming** Pane (for Naming Guidelines Only)

If the **Call Graph** pane has focus, the **View by** list will be visible. This list lets you filter which methods/properties are listed on the Solution Tree. You can choose from the following:

◆ **All**: To list all methods or properties in the current solution

◆ **Active**: To list only the active (live) methods or properties in the treeview

◆ **Uncalled**: To list only the uncalled methods or properties in the treeview

Note: See "Call Graph References" on page 63 for information about active and uncalled references. Consult the DevPartner Code Review online help on how to use the **View by** list for the **Call Graph** pane.
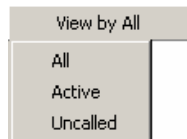
Figure 3-20. **View by** List for **Call Graph** Pane

Naming Analysis Functionality

The DevPartner code review feature incorporates two kinds of naming analysis capabilities:

◆ Naming Guidelines

The naming analyzer targets support for the .NET Framework. See "Naming Guidelines Naming Analyzer" on page 58.

◆ Hungarian

The Hungarian naming analyzer remains a legacy naming analyzer in the DevPartner code review feature. See "Hungarian Naming Analyzer" on page 61.

Note: You can also choose **None** from the **Naming analysis to use** list on the General Options property page (Figure 3-14 on page 54). However, doing so tells DevPartner to bypass naming analysis altogether. The **Naming** pane will be empty following the review.

Naming Guidelines Naming Analyzer

The Naming Guidelines naming analyzer is patterned after the Visual Studio .NET Framework naming guidelines. Microsoft adopted the Visual Studio naming guidelines to support the .NET Framework. These naming

guidelines ensure that consistent, predictable, and manageable naming practices are applied to .NET Framework types in a managed class library.

Note: To perform this type of naming analysis, choose **Naming Guidelines** from the **Naming analysis to use** list on the General Options property page (Figure 3-14 on page 54), plus make additional selections on the Naming Guidelines property page (Figure 3-15 on page 55) to ensure a more precise review. Consult the DevPartner Code Review online help on how to make selections.

The Naming Guidelines naming analyzer examines parameters, classes, namespaces, methods, delegates, enums, structs, interfaces, and variables. It looks for naming violations in the source code related to capitalization, case sensitivity, abbreviations and acronyms, and syntax for namespaces and other .NET Framework identifiers. The following sections describe some guidelines with which the Naming Guidelines naming analyzer complies.

Note: Consult the DevPartner Code Review online help for additional information regarding these guidelines.

Capitalization

When it finds a naming violation, DevPartner attempts to suggest a more appropriate name on the **Naming** pane using the capitalization style that you selected on the Naming Guidelines property page — Camel or Pascal.

Table 3-14. Capitalization Styles Used in Naming Guidelines Naming Analyzer

Capitalization style	First concatenated word	Subsequent concatenated words	Examples of Suggested Names
Camel case	Not initial-capped	Initial-capped	**red**C**olor
Pascal case	Initial-capped	Initial-capped	**R**ed**C**olor

Case Sensitivity

DevPartner discourages using case sensitivity to differentiate identifiers in the source code. Case *insensitivity* is strongly encouraged because it supports interoperation between case-sensitive and case-insensitive programming languages and also reduces confusion between two similarly-named identifiers. Developers should avoid names that vary only by case. Rather, they should use names that are functional in either case-sensitive or case-insensitive programming languages.

Abbreviations and Acronyms

DevPartner supports the use of generally accepted abbreviations and acronyms. DevPartner determines proper naming based on the number of letters for the abbreviation or acronym and where it is situated in the identifier name.

Namespace Syntax

DevPartner supports the .NET Framework naming convention for namespaces. That is, that the namespace name starts with the company name, followed by the technology name, and optionally ending with the feature and/or design name. Here is an example of the established syntax:

```
CompanyName.TechnologyName[.Feature][.Design]
```

By default, DevPartner recommends Pascal case for namespaces. The period character (.) separates each logical concatenated word. You enter the namespace information in the **Namespace options** field on the Naming Guidelines property page (Figure 3-15 on page 55) prior to the review.

Syntax for Other .NET Framework Identifiers

DevPartner checks that .NET Framework identifiers are properly named in the source code. Here are some examples of what DevPartner looks for:

◆ Numeric characters

DevPartner checks whether numbers are part of the identifier name. While DevPartner will not remove the numeric characters, it will flag the name as a violation.

◆ Underscore characters

DevPartner looks for instances of the underscore character (_) in the identifier name. The underscore character is discouraged in the Naming Guidelines naming analyzer. DevPartner will remove the underscore character except in the following cases:

◇ If the underscore is a leading character (i.e., _redColor)
◇ If it is used in a method name
◇ If its removal will introduce another naming violation

◆ Casing for constants

DevPartner follows Pascal or Camel casing for constants (depending on the case selection you made on the Naming Guidelines property

page), rather than all uppercase. For example, DevPartner would change the constant HTTP_PORT in:

```
private const int HTTP_PORT = 80
```

◇ To HttpPort based on Pascal
◇ To httpPort based on Camel

◆ Delegate

If a delegate identifier name includes the word delegate (regardless of case) along with one or more identifiable words, DevPartner will remove the word delegate as long as it does not introduce another violation. For example, the name, MyDelegateWord, would be renamed as MyWord.

Hungarian Naming Analyzer

DevPartner continues to include the Hungarian naming analyzer, patterned after the Hungarian Notation naming convention.

Note: The Hungarian naming analyzer has always been part of the DevPartner code review feature. In this release, however, it has been named *Hungarian* to differentiate it from the naming analyzer, *Naming Guidelines*. To perform this type of naming analysis, choose **Hungarian** from the **Naming analysis to use** list on the General Options property page (Figure 3-14 on page 54).

Based on the Hungarian naming convention, variable names include specific character(s) that identify a particular scope-level or data-type prefix for the variable in question. For example, the data-type prefix int signifies an integer, such as integer variable Port; and the scope-level prefix g_ signifies global, as in g_intPort.

DevPartner uses the Hungarian naming analyzer in a code review when the **Hungarian** option is selected from the **Naming analysis to use** list on the General Options property page (Figure 3-14 on page 54). DevPartner also uses the currently selected name set. When a code review is started, the naming analyzer evaluates scope-level prefixes and data-type prefixes for every variable in the code. If applicable, it makes recommendations consistent with the name set (*Default* preferred) and displays the naming results on the **Naming** pane (Figure 3-9 on page 48) following the code review.

Note: The Hungarian naming analyzer does not evaluate parameter names.

The following tables list examples of scope-level and data-type prefix combinations that are evaluated in the Hungarian naming analyzer, as specified in the current name set:

Table 3-15. Scope Prefix

Scope	Prefix
Global	g_
Member	m_
Local	""

Table 3-16. Data Type Prefix

Data type	Prefix
string	str
int	int
int	i
boolean	bool
bool	bln

The qualifiers on a variable declaration determine the scope, such as the boundaries where the variable exists. For example, DevPartner considers a variable with public status as having a global scope because it is accessible outside the class.

The default name set contains scope prefixes that can be edited using the Rule Manager. You can also customize variable and object names, based on Hungarian Notation, using the Rule Manager.

Note: Consult the DevPartner Code Review online help for more information on the Hungarian naming analyzer and how DevPartner constructs more suitable naming recommendations. Consult the DevPartner Code Review Rule Manager for more information on managing name sets.

Call Graph Analysis Functionality

The **Call Graph** pane (Figure 3-13 on page 53) shows a graphical representation of the call graph data collected from the review. It displays a static view of the inbound and outbound call path corresponding to the method or property selected from the Solution Tree.

Note: Call paths are statically generated, not dynamically. This means that the graph shows the potential method calls in the call path, rather than the actual calls made during a program's execution.

Why the Call Graph Pane Is Initially Empty

The Call Graph pane will be empty if:

◆ The **Collect call graph data** check box on the General Options property page (Figure 3-14 on page 54) was not selected prior to the code review.

Call graph data was not collected during the review. To ensure call graph analysis is performed, select this option and then perform another code review.

◆ You did select the check box, but you did not click a method or property on the Solution Tree.

Data was collected but no call graph appears by default. To view a call graph in the display area, click a method or property node on the Solution Tree.

Call Graph References

The **Call Graph** pane depicts the potential inbound/outbound call references in a call path. It traces the call hierarchy for the selected method or property. The contents of the display area show the potential entry and exit points for each method or property. The call references start at the root node with all calls performed in reference to it until control is returned to the root node or completed from the root node. The following types of call references appear in the display area:

Root Node   The root node refers to the method or property selected to be the starting point of the call graph. All other nodes either call into the root node or are called by it. The root node (Figure 3-21) appears as a light yellow rectangle with a wide blue border, to distinguish it from all other nodes in the display area.
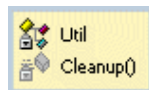
Figure 3-21. Example of Root Node

Inbound Calls   Inbound refers to methods or properties that directly or indirectly call into the root node. The inbound calls (Figure 3-22) are shown as light blue rectangular nodes, to differentiate them from the root node.

Outbound Calls

Outbound refers to methods or properties that are directly or indirectly called by the root node. As with the inbound calls, the outbound calls (Figure 3-22) appear as light blue rectangular nodes. They are connected by a series of arrows, pointing away from the root, to show the potential direction of the call path.
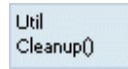
Figure 3-22. Example of Inbound or Outbound Call Node

Uncalled References

Uncalled refers to a method or property that is defined in the code but never referenced within the files that form an application component. Uncalled methods are identified on a node on the **Call Graph** pane, either by the label **Uncalled** or the symbol (!).

 with label

 with icon
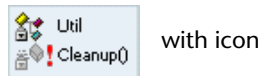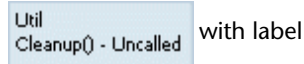
Figure 3-23. Two Examples of Uncalled Identification

Recursive and Circular Call References

The **Call Graph** pane can graphically show instances of recursive or circular call references that exist in the selected path of execution.

◆ Recursive: Method or property that calls itself in the path of execution

A calls B;
B calls B

Figure 3-24. Example of Recursive Call Graph

◆ Circular: Method or property that indirectly calls back into a previously called method or property in the path of execution

A calls B;
B calls C;
C calls back to A
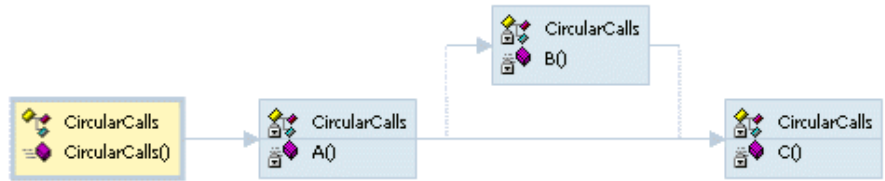
Figure 3-25. Example of Circular Call Graph

Call Graph Configuration Options

DevPartner provides four ways to configure how a call graph appears on the **Call Graph** pane. You can access these options either from the Call Graph toolbar (Figure 3-18 on page 56) or by right-clicking on the background area of the **Call Graph** pane.

Number of Levels

You can choose the number of levels to be displayed on the **Call Graph** pane. The call graph shows a specified number of levels of methods or properties that call into (inbound) and are called from (outbound) the root node. You can choose between one and six levels (six, default). The following example shows two levels selected. The plus signs (+) on the nodes to the right of the call graph indicate that more levels of call references are available for viewing.

Figure 3-26. Shows Two-level Configuration
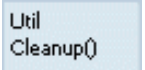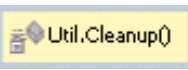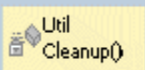
Node Style

You can choose the node style that will be applied to the **Call Graph** pane. All call graph node styles show the class name, as well as the method or property name. Some node styles also include icons indicating the access type of the class, method, or property: public, private, internal, or protected. These are standard Solution Tree icons. Other icons, representing uncalled methods and properties, only appear in the call graph.

The following table shows examples of the various node styles.

Note: Some examples show root node and others, standard (inbound or outbound). See "Call Graph References" on page 63 for more information on how nodes are differentiated.

Table 3-17. Node Styles

Node Style	Description	Uncalled Representation	Examples
Single label	Shows the class name, then a period, followed by the method or property name, but without icons	The string, - Uncalled, will append the method or property name.	Util.Cleanup() Inbound/outbound
Top and bottom labels	Shows the class name appearing on the first line and the method or property name appearing on the next, but without icons	The string, - Uncalled, will append the method or property name on the second line.	Util Cleanup() Inbound/outbound
One image and label	Shows a standard method or property icon, plus the class name, then a period, followed by the method or property name, all on the same line	The corresponding icon will include an exclamation point icon (!).	Util.Cleanup() Root
One image and two labels	Shows an icon for the method or property, along with the class name on the first line and the method or property name on the second line	The corresponding icon will include an exclamation point icon (!).	Util Cleanup() Root
Two images and two labels	Shows an upper-level icon for the class followed by the class name, and a lower-level icon for the method or class, followed by its name	The explanation point icon (!) icon will appear between the data type icon and the name.	Util Cleanup() Root

Scaling   You can choose the relative size of the call graph on the **Call Graph** pane. Two scaling options are available:

◆ To fit in available space (default)

This selection lets you scale the call graph so that all the nodes fit within the display area. By default, scroll bars are not available with this choice. If you reconfigure the call graph using the other options, the contents will be resized, without the inclusion of scroll bars.

◆ By percent of full size

This selection lets you enlarge or shrink the contents in the display area by one of these fixed percentage values: 100%, 80%, 75%, 66%,

or 50%. This choice allows you to zoom into sections of a large or complicated call sequence. Moreover, when the contents are redrawn, the selected method or property (root node) is clearly visible in the display area. Scroll bars are also available.

Layout You can choose how the call graph nodes will be laid out on the **Call Graph** pane. Your choices include:

◆ Horizontal

The nodes will appear in a left-to-right orientation in the display area. The methods or properties calling into the selected node (also called the root node) will be located to its left. The methods or properties that the selected node calls into will branch to the right.

◆ Vertical

The nodes will appear in a top-to-bottom orientation in the display area. The methods or properties calling into the selected root node will be located above the root node. The methods or properties that the selected node calls into will be located below it.

DevPartner Code Review Rule Manager

DevPartner contains an extensible rules database that is based on the Microsoft Visual Studio programming standards. The rules database is maintained and stored in the Rule Manager. The Rule Manager is a standalone application that accompanies the DevPartner code review feature. You can access it from **Compuware DevPartner Studio > Utilities** from the **Start** menu. When selected, its user interface opens in a separate window.
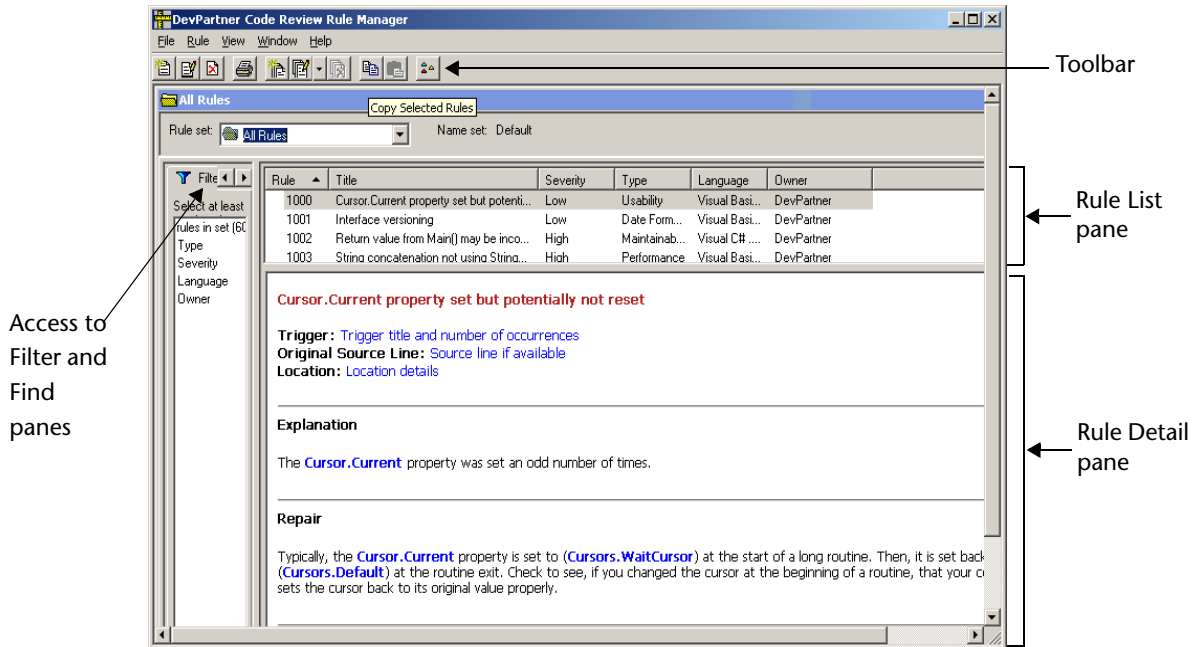
Figure 3-27. **DevPartner Code Review Rule Manager** Window

Note: Consult the DevPartner Code Review Rule Manager online help for information on Rule Manager functionality.

Creating New Rules Using Regular Expressions

Developers can create their own powerful rules in DevPartner Code Review, and use them to identify many suspect coding practices. Code Review rules make extensive use of regular expressions, which provide a robust and versatile method for searching text. Regular expressions are widely used, well-documented, and can be written to match patterns in HTML, VB.NET, and C# syntax. DevPartner Code Review uses the same regular expression engine as Microsoft Visual Studio, and supports the same syntax.

Code Review makes it easier to use regular expressions in its rules by limiting the scope of any given rule to certain parts of the code. For example, a rule can apply to the entire file, just methods, or only While blocks. Since rules can specify a scope, the regular expressions can focus on a targeted part of the code.

Code Review also assists the regular expression search by removing comments from a code block before executing the review, reducing false positives.

The following sections provide examples of actual Code Review rules and explain the regular expressions that drive them.

Note: To learn more about how to write regular expressions for your Code Review rules, read the following books, or visit the following Web sites:

◇ Forta, Ben. *Teach Yourself Regular Expressions in 10 Minutes*. Indiana: Sams Publishing, 2004.

◇ Friedl, Jeffrey E.F. *Mastering Regular Expressions*. 2nd ed. California: O'Reilly, 2002.

◇ Goyvaerts, Jan. *Regex Tutorial, Examples and Reference*. 1 Feb. 2006 <http://www.regular-expressions.info>.

◇ Microsoft Corporation. *.NET Framework Regular Expressions*. 2006. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcomregularexpressions.asp>

Matching Lines Exceeding 90 Characters

Best practice coding standards recommend that a line of code should not exceed 90 characters. A Code Review rule enforces this standard by searching for lines that exceed 90 characters. The following regular expression ensures lines do not exceed 90 characters in length:

```
(?-s).{91,}
```

This regular expression first sets the **Single Line** option to **False**, causing the expression to evaluate all characters up to, but not including, the newline character (\n) as a single line. This evaluation treats each line of code, from its beginning to the newline (\n) character, as a distinct and different line.

Next, the rule incorporates the most elementary aspect of regular expressions — matching single characters. This rule uses the period (.) metacharacter to match any single character on the line.

The rule follows the period (.) with a repeating match metacharacter {91,}. Repeating match metacharacters specify that a match must repeat a certain number of times, or within a certain range of instances. In this rule, it specifies the expression is true only if any single character is matched 91 or more times; the second value in the range is left empty, because the rule only cares if the number of matches exceeds 90 characters. Table 3-18 describes the basic repeating match metacharacters.

Table 3-18. Repeating Match Metacharacters

Character	Meaning
+	Matches one or more instances of the preceding character
*	Matches zero or more instances of the preceding character
?	Matches zero or one instance of the preceding character
{*n*} {2,6} {*n*,}	Matches an exact number of instances of the preceding character, where *n* represents the number of required repetitions These braces are also used to designate a range of repetition, such as from two to six times, by including the upper and lower limit separated by a comma. Omitting the upper limit matches against a minimum number of instances, without an upper bound.

Matching Tabs Used Instead Of Spaces

Best practice coding standards recommend that spaces be used instead of tabs. The number of spaces represented by a tab can differ between editors, and this difference can cause the source code to have a different appearance in each editor. To enforce a consistent appearance of the source, spaces should be used. The following regular expression is used in a Code Review rule that searches for the use of tabs inside of methods:

```
(?s)\t.*
```

This regular expression sets the **Single Line** option to **True**, causing the expression to evaluate every character on every line, up to and including newline characters (\n), as part of a single line.

Next, the rule specifies a match against the tab character by using a metacharacter (\t). Without further change, the regular expression would find every occurence of a tab character in the method. Instances where multiple tabs are used in a method, such as for indenting lines, would fire the rule for each tab in that method. That is not the intended behavior of the rule.

This rule should evaluate to true if at least one tab is used in the method, but not every time it encounters a tab in the method. To accomplish this result, the rule needs the period (.) metacharacter followed by a repeating match metacharacter specifying zero or more instances. Table 3-18 shows the repeating match metacharacter to use is the asterisk (*). Adding these last two metacharacters specifies that the rule will evaluate to true the first time it encounters a tab character, and then capture every following character in the method.

Matching Instances Where Code Catches System.Exception

Avoid catching `System.Exception` to handle your errors, as it does not catch errors at a fine enough level of detail to allow for the proper differentiation of error types. Error handling code blocks should intercept and handle errors at as fine a granularity as possible, since doing so can make a program more robust and less likely to crash. The following regular expression is used in a Code Review rule designed to find instances in the code where Visual Basic .NET syntax is used to catch `System.Exception`:

```
Catch\s\w+\sAs\s(System\.)?Exception
```

The first part of this expression locates any instance of the literal word `Catch` in the code. Since the rule should not match instances where Catch is the first part of a longer word, the literal text is followed by the metacharacter for whitespace (`\s`).

Visual Basic .NET syntax uses the word `Catch` followed by a variable name (used to hold the exception object). The variable is followed by more whitespace, and the literal word `As`.

The rule needs regular expression functionality to locate a legal variable name, followed by more whitespace, and the word As. The metacharacter `\w`, paired with the repeating match metacharacter `+`, will locate one or more instance of any alphanumeric character (upper or lower case) or the underscore character. Adding `\sAs\s` finishes the search for a legal variable name followed by whitespace and the word As.

So far, the regular expression will locate the following code:

```
Catch MyExceptionObject As
```

This regular expression would do a great job of locating all code that is catching exceptions. However, the rule should only match against code that catches `System.Exception`. It requires further refinement.

To ensure that it only matches instances where the code catches `System.Exception`, the regular expression searches for the literal words `System` and `Exception` separated by a period. Since the period is a metacharacter, the rule needs to specify a match on a literal period by preceding it with the backslash, removing its special character status.

If the rule now has `System\.Exception` as part of the regular expression, there is still a problem. It is perfectly acceptable syntax for the catch of `System.Exception` to leave off the `System.` and only use the term `Exception`. One last modification to the regular expression makes the matching of `System.` optional. Wrapping `System\.` in parenthesis makes it a subexpression, which can be followed by the `?` metacharacter to specify zero or one match.

Matching Methods Having More Than One Return Point

Best practice coding standards recommend that methods have only one return point. Having more then one return point could cause code to be hard to understand. The following regular expression is used in a Code Review rule that locates instances where a method has more than one return point. Most of the pieces making up this expression have been used in previous rules, but there are a couple of new things to examine.

```
(?s)(\breturn\b.*){2,}
```

First the rule sets the **Single Line** option to **True**, using `(?s)`, to focus on the entire method. To consider the method as a whole, the rule needs to evaluate every character on every line, up to and including newline characters (`\n`), as part of a single line.

Another part of the expression used in earlier rules is the repeating match metacharacter at the end. This expression uses `{2,}` to modify the preceding subexpression (contained in parenthesis), requiring that there must be two or more matches within the method.

The subexpression, `(\breturn\b.*)`, is the part of the regular expression doing most of the work. It is written as a subexpression to allow the whole block to be modified by the repeating match metacharacter. The metacharacter `\b` is a word boundary. By surrounding the literal text `return` with the word boundary metacharacters, the regular expression will look for instances of the word standing alone, not as part of a larger word.

Note: The last rule followed the literal text `Catch` by the whitespace metacharacter `\s` to ensure it would only find instances where Catch was a whole word. This is a good example of how flexible regular expressions are. That rule could have used the word boundary metacharacter `\b`, but did not.

The final `.*` within the subexpression searches for a match of zero or more instances of any character. The rule is now complete, and will search entire methods for two or more instances of the word `return`, followed by zero or more characters.

Enforcing Initialization Of Variables When They Are Defined

As a best practice, to keep code concise and easy to understand, variables should always be initialized when they are defined. The following regular expression is from a Code Review rule that locates instances where a variable is defined, but not initialized:

```
(?-s)\bDim\b(?!.*=)(?!.*\bnew\b)
```

Since the rule needs to evaluate each line of code by itself, the first thing it does is set the **Single Line** option to **False**.

Next, the regular expression is going to look for the word `Dim`. It wraps the literal text `Dim` with word boundary metacharacters `\b` to ensure it is only considering whole words.

The subexpression implements the concept of looking ahead or behind. The ability for regular expressions to look ahead or behind gives them additional flexibility.

Looking ahead or behind means that the regular expression is searching for a match in the text, but instead of matching that text itself, it allows the rest of the regular expression to succeed if the subexpression succeeds. This kind of looking ahead and behind is referred to as a *positive look ahead* or *positive look behind*, because the expression is searching for text that will match.

The syntax for the positive look ahead and positive look behind is:

◆ Positive look ahead `(?=`*subexpression*`)`
◆ Positive look behind `(?<=`*subexpression*`)`

Similarly, *negative look ahead* and *negative look behind* work by searching for text that does *not* match the subexpression specified in the statement.

The syntax for the negative look ahead and negative look behind is:

◆ Negative look ahead `(?!`*subexpression*`)`
◆ Negative look behind `(?<!`*subexpression*`)`

The regular expression for this rule needs to use the negative look ahead construct to detect when there is *not* an equal sign (`=`) to the right of the `Dim` keyword, with or without a space preceding it. The subexpression `(?!.*=)` handles that negative look ahead.

The last part of the expression, `(?!.*\bnew\b)`, uses another negative look ahead to evaluate to true if the word `new` does *not* exist to the right of the `Dim` keyword, with or without a preceding space.

The complete rule now has a regular expression that will evaluate to true any time it encounters a line of code where the word `Dim` is not followed by an equal sign (`=`) or the word `new`.

Matching Instances Of More Than One Statement Per Line

In order to increase readability and maintainability of code, more than one statement should never be placed on a single line (with the exception of loop syntax). The following regular expression is from a Code Review rule that locates instances where a line contains more than one statement:

```
(?<!for.*);.*;
```

It might appear that the easiest way to detect more than one statement on a given line would be to determine if any line contains more than one semicolon (;). In fact, this search is the essence of the regular expression in this rule, but it needs to also take into account the possibility of a semicolon being associated with the `for` keyword.

To exclude any instances where the keyword `for` is associated with a semicolon, the rule uses the negative look behind construct (?<!for.*) to look back on any line where it encounters a semicolon, making sure the word `for` is not there. The remaining part of this regular expression (;.*;) will search for a semicolon followed by any number of other characters, and then another semicolon.

Ensuring Open Braces Are Placed On A Separate Line

Best practice coding standards recommend that open braces should be placed at the beginning of their own separate line following the statement that begins the block. The following regular expression is from a Code Review rule that locates instances where open braces are not placed on their own separate line:

```
(?m)^\s*\w+(?=.*?\{).*?$
```

There are several new concepts at work in this expression. The rule first sets the **Multi Line** option to **True** with (?m). This setting changes the behavior of two other metacharacters — **Line Beginning** (^) and **Line End** ($). By enabling the multi line option, ^ and $ capture the beginning and end of each line rather than the entire string being searched.

Once multi line mode is enabled, the regular expression searches for the beginning of the line (^), followed by one or more whitespace characters (\s*), and one or more word characters (\w+). This sets up the basis the regular expression will use, because if it finds one or more word characters there should be no open braces on the line.

The positive look ahead subexpression `(?=.*?\{)` searches through each line looking for any character followed by an open brace. The backslash before the open brace removes its metacharacter status. Once the rule determines that a line contains a character followed by an open brace, and the open brace is not on a line by itself, `.*?` at the end of the regular expression allows it to capture the remaining text right to the end of the line (matched by the `$` metacharacter).

Ensuring Loop Counters Are Not Modified Inside the Loop Bodies

Changing the loop counter inside the body of the loop could cause unpredictable results, and always makes code harder to understand. The following regular expression is from a Code Review rule that locates any instance where a loop counter is modified inside of the loop body:

```
(?s)\bfor\b\s*\(\s*\w+\s+(?<VARNAME>\w+).*\).*\b\k<VARNAME>\b\s*=
```

This regular expression is extremely long because it has to do a lot of work to enforce the rule. To identify and store the loop counter variable name, the regular expression must first capture the `for` keyword, left parenthesis, and loop counter type.

The first half of the regular expression is gathering required information: it locates a line with the `for` keyword, followed by any number of whitespace characters, a left parenthesis, more whitespace characters, one or more word characters, more whitespace, and finally uses a subexpression to capture the variable name.

The subexpression `(?<VARNAME>\w+)` will capture the name of the loop counter and store it in the variable, VARNAME.

Note: This construct can use any variable name, as long as the name does not contain any punctuation and does not begin with a number.

Once it has captured the name of the loop counter, the last part of the regular expression captures any remaining characters and the right parenthesis. It then begins searching through the loop body for an instance of the loop counter, followed by an equal sign, using the following construct:

```
.*\b\k<VARNAME>\b\s*=
```

This part of the expression is the essence of the rule. Prior to reaching this point, the regular expression has determined the name of the loop counter and has placed the scope of its search within the loop body. This final part of the expression now matches all characters up to the value of VARNAME (the loop counter), and then looks for an equal sign (=) following the counter.

The fact that the loop counter is followed by an equal sign indicates it is being set to some value, or modified. Since the counter should never be modified inside of the loop body, the rule has found a violation.

Chapter 4

Automatic Code Coverage Analysis
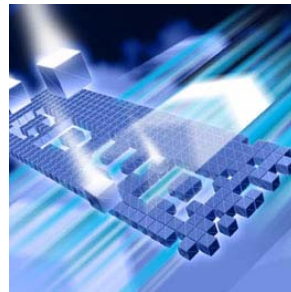
Introducing DevPartner Coverage Analysis

DevPartner includes a code coverage analysis feature that lets developers and testers automatically locate untested code in software applications and components developed in Visual Studio. DevPartner can collect coverage data for managed code applications, including Web and ASP.NET applications, as well as unmanaged (native) Visual C++ and Visual Basic applications.

DevPartner can also collect coverage data for Visual C++ and Visual Basic 6 applications or components. See "Coverage Analysis for Visual Basic and Visual C++" on page 80.

DevPartner coverage analysis helps development teams save testing time and improve code reliability by measuring and tracking code execution and code base stability during development.

DevPartner allows you to gather coverage data without leaving Visual Studio. You can collect coverage data for a managed or unmanaged application any time you run the application from the **DevPartner** menu in Visual Studio. DevPartner also lets you analyze your applications and components as they are really used — as native applications, ADO.NET components, or ASP.NET or Web applications. DevPartner can even collect data for managed code applications started outside of Visual Studio.

What is DevPartner Coverage Analysis

DevPartner coverage analysis provides these essential coverage analysis features:

◆ Data collecting and reporting
◆ Focused data collection
◆ Data correlation for distributed applications
◆ Data merging for single-process applications
◆ Multi-language code coverage analysis
◆ Integration with DevPartner Error Detection
◆ Export DevPartner Data

Data Collecting and Reporting

DevPartner coverage analysis gathers coverage data for applications, components, images, methods, functions, modules, and individual lines of code.

DevPartner gathers coverage data for executable images, executable server extension code (such as an ISAPI DLL), and scripts contained in HTML files and written in the scripting languages supported by Internet Explorer and Microsoft's Internet Information Server (IIS). The scripting languages can also invoke COM objects that can be in-process or out-of-process.

Focused Data Collection

DevPartner session controls let you focus your coverage analysis on any phase of your application. You can use the session controls to stop data collection, clear data collected to that point, or take a snapshot of the data currently collected and then continue recording.

Data Correlation for Distributed Applications

DevPartner coverage analysis can collect session data for multiple processes participating in a single run of a multi-tier application. To

collect coverage data on applications that include both client and server components, you use DevPartner in Visual Studio. If the server component runs on a remote system, you install and configure DevPartner on the remote system as well.

For example, you can install DevPartner on the server to collect data for IIS, and use coverage analysis in Visual Studio to collect data on your client-side application. As you use your application and access components on the server, DevPartner collects coverage information on both the server and the client.

To collect data simultaneously from both machines, install DevPartner on the client and DevPartner and the optional DevPartner Server license on the remote machine. See *Installing DevPartner* (DPS Install.pdf) and the *Distributed Licensing Management License Installation Guide* (LicInst4.pdf) for more information.

When you take a data snapshot on the client, or exit from your application, DevPartner automatically creates a session data file that contains correlated coverage data for both the client and the server portions of your application. When you view the data, you see an integrated view of the coverage of your entire application.

Data Merging for Single Process Applications

You can accumulate data by running the single-process application or component more than once and collecting the data into multiple session files, which you can then merge. Merging combines the coverage data from multiple session files into a single file. Accumulating data lets you track changes in your code so that you can gauge the stability of your code base. If several members of your development team use DevPartner, you can merge the files generated by all the team members to determine project-wide coverage statistics.

Note: You cannot merge correlated session files or Web script session files produced from running Internet Explorer. You can merge server-side session files from Internet Information Server.

Multi-Language Code Coverage Analysis

DevPartner supports all Visual Studio managed code languages, as well as native C/C++. DevPartner can also collect coverage data for Visual Basic and Visual C++ 6.0 applications, as well as JScript and VBScript Web applications when using Internet Explorer or IIS.

Coverage Analysis for Visual Basic and Visual C++

DevPartner 8.1 provides limited integration for Visual Basic 6.0 and Visual C++ 6.0. You can build your Visual Basic and Visual C++ applications with coverage instrumentation in the Visual Studio 6.0 environment.

The sections "Analyzing Coverage in Visual C++" on page 98 and "Analyzing Coverage in Visual Basic" on page 99 provide an overview of the procedure and direct you to other sources of information.

Integration with DevPartner Error Detection

You can use DevPartner error detection with coverage analysis to collect coverage data and check for errors during the same session when you run your managed code application or native C/C++ application in the debugger. You must instrument native C/C++ applications for Error Detection and Coverage with the **Native C/C++ Instrumentation Manager** before collecting data.

Export DevPartner Data

The Export DevPartner Data feature enables you to export DevPartner coverage session files (with the `.dpcov` extension) and merged coverage files (with the `.dpmrg` extension) to XML. When this feature is active, the **Export DevPartner Data** command is available on the **File** menu.

You can analyze the data in the exported XML file using your own or third-party software. For example:

◆ Use Export DevPartner Data on a development build server or QA server where unit tests, functional tests, or regression tests are staged. Analyze the exported XML data to monitor daily progress.

◆ Use Export DevPartner Data to collect data for longer-term analysis. You can accumulate the XML data in a database or data warehouse in order to:
   ◇ Integrate the data with development and QA methodologies, tools and infrastructure
   ◇ Run custom analytics on the data
   ◇ Archive the data for historical or auditing purposes

How DevPartner Coverage Analysis Fits in Your Development Cycle

You can run DevPartner coverage analysis sessions during the development and testing phases of the software development cycle. Developers use DevPartner prior to significant milestones (code check-in,

unit testing, integration, internal release) and when the code base for an application or component becomes relatively stable to ensure that the desired level of testing or execution has been reached.

Software testers can also use DevPartner during routine regression testing and reliability testing to ensure that applications and components have been thoroughly exercised under test conditions prior to release or deployment.

DevPartner Support for Visual Studio

This section provides you with information on how the DevPartner coverage analysis feature integrates into Visual Studio.

DevPartner IDE Integration in Visual Studio

DevPartner is fully integrated into Visual Studio. This makes it easy to collect code coverage data for an application regularly, as you develop it, without leaving the development environment. Once DevPartner is enabled, you collect coverage data by running your application as you run any application you are developing, by starting it from the **DevPartner** menu.

When the analysis session completes, DevPartner displays the coverage data in the IDE. DevPartner saves the coverage data in a coverage analysis report file, with a `.dpcov` extension. Coverage files are automatically added to the **DevPartner Studio folder** in the active solution. To review an existing coverage analysis file, double-click the file in Solution Explorer.

DevPartner Toolbar and Menu Integration

DevPartner adds commands related to coverage analysis to several Visual Studio menus. In addition, DevPartner provides toolbars that include shortcuts to basic coverage analysis functions.

Figure 4-1. The DevPartner toolbar

◆ **DevPartner toolbar**
  ◇ Starts coverage data collection. Menu equivalent: **Start with Coverage Analysis** on the **DevPartner** menu.

◇ 🔧 Starts coverage data collection for an error detection session. Available for unmanaged (native) C/C++ applications. Menu equivalent: **Start with Error Detection and Coverage Analysis** on the **DevPartner** menu.

◇ 🔧 Enables native C/C++ data collection when you rebuild the solution or project. Menu equivalent: **Native C/C++ Instrumentation** on the **DevPartner** menu.

◇ 🔧 Opens the DevPartner options pages. Menu equivalent: **Options** on the **DevPartner** menu.

Figure 4-2. The DevPartner Session Controls toolbar

◆ **DevPartner Session Controls toolbar**

◇ ■ **Stops** data collection and takes a final data snapshot

◇ 🔧 Takes a data **Snapshot**

◇ ✕ **Clears** data collected to the point at which the Clear action executes

◇ Process list focuses data collection on a single process for applications that run in multiple processes.

◆ **DevPartner Menu**

◇ **Start with Coverage Analysis** Enables coverage data collection

◇ **Start with Error Detection and Coverage Analysis** Enables simultaneous error detection and coverage data collection

◇ **Native C/C++ Instrumentation** Activates DevPartner instrumentation.

◇ **Native C/C++ Instrumentation Manager** Opens the DevPartner Native C/C++ dialog box which enables you to set DevPartner instrumentation options for unmanaged (native) C/C++ projects before rebuilding.

◇ **Correlate > Coverage Files...** Combines client and server-side session files into a single correlated file.

◇ **Merge Coverage Files...** Merges individual session files into a merge file.

◇ **Options** Accesses the Analysis options pages to configure coverage data collection.

See the *Coverage Analysis* section in the DevPartner online help in the Visual Studio .NET Combined Collection for information about other commands DevPartner adds to the Visual Studio menus and context menus, and how existing Visual Studio menu commands relate to the DevPartner coverage analysis feature.

Solution Explorer Integration

When you run a coverage analysis session from Visual Studio, DevPartner adds the resulting session file or files to the **DevPartner Studio** folder in Solution Explorer.

From Solution Explorer you can open a session file by double-clicking, or use the file or folder context menu options to add or remove files from the solution, view properties, and cut or copy the file specification to the clipboard.

Collecting Coverage Data

This section provides information about using DevPartner coverage analysis to collect data from different types of applications.

Running Your Program under DevPartner Coverage Analysis

This section describes how to instrument and run your application with DevPartner to collect coverage analysis data.

For Managed Code Applications

Many applications you will develop in Visual Studio will be managed code applications. C#, Visual Basic .NET, and managed Visual C++ applications are examples of managed code applications. In order to analyze an application as it runs, DevPartner instruments the application code for the specified type of data collection, in this instance, coverage data. In the case of managed code applications, DevPartner instruments for data collection at runtime, as the application code is compiled for execution by the common language runtime.

As a result, it is easy to collect data for managed code applications.

1   In the Visual Studio **Properties** window, review the **DevPartner Coverage, Memory and Performance Analysis** properties for the solution and change as needed.

◇   To display the **Properties** window, select **View**, then choose **Properties Window**.

2   Click **Start with Coverage Analysis** on the DevPartner toolbar.

3   Exercise the application. You can use the session controls, described on , to focus data collection.

4   Exit the application.

DevPartner collects coverage data as the application runs, and displays the data in the main window in Visual Studio when the analysis session ends. The session data file (`.dpcov`) appears in the **DevPartner Studio** folder in Solution Explorer. If you used the Snapshot session control, DevPartner creates a session file for each snapshot.

For Unmanaged (Native) Visual C++ Applications

Unlike managed code, which DevPartner instruments at runtime, you must instrument unmanaged (native) C/C++ code when you compile it. DevPartner makes this as easy as rebuilding your solution or project.

Choose **Native C/C++ Instrumentation Manager...** on the **DevPartner** menu. In the **Instrumentation Manager**, choose the type of instrumentation and select the native C/C++ projects you want DevPartner to instrument. Then, rebuild the solution, or rebuild the specific projects.

Once your unmanaged C++ application or project is instrumented, run the application as described above.

If your Visual Studio application includes managed and unmanaged portions, DevPartner collects data for both, provided the managed and unmanaged portions are in separate files.

For Web Applications

If you develop Web Forms, XML Web Services, or ASP.NET applications, you can use DevPartner to collect coverage data for both client and server portions of your application. If you start your application from Visual Studio, you enable coverage analysis and run the application. DevPartner collects the coverage data as it would for any other managed code application.

However, there are several things you need to be aware of when analyzing Web applications.

Note: If IIS runs on the local machine, set the following options on the system. If IIS runs on a remote server, you must install DevPartner (and a Server license) on that system and configure it for data collection.

Before you begin collecting data for analysis:

◇ *Warm up* the application by exercising it for several minutes. Be sure to include the parts of the application in which you are interested.

◇ Execute the **Clear** session control action to discard data collected to that point.

◇ Collect data.

In this way, you can eliminate data collection for the many one-time initializations that take place when you launch the application.

Collecting Data from Multiple Processes

Web applications may run more than one process. For example, when you profile an ASP.NET application you may see the browser process (`iexplore`), the IIS process (`inetinfo`), and the ASP worker (`aspnet_wp` or `w3wp`) processes.

When you run such an application under coverage analysis, the DevPartner **Session Control** toolbar displays the active processes in the process selection list. Use the process list to focus data collection. When you execute a Snapshot session control action, DevPartner creates a snapshot session file with data for the process selected in the process list.

Note: The process list in the DevPartner Session Control toolbar includes all active processes in the analysis session. However, if all processes run on the local machine, DevPartner launches a separate version of the Session Control toolbar for each process. These instances of the toolbar reflect only a single process. You can use the separate toolbar to execute session control actions for that process, or use the primary Session Control toolbar in Visual Studio to select any active process and execute session control actions.

Collecting Server-side Coverage Data

You may want to collect coverage data for both client and server portions of a client/server application. With DevPartner, you can collect coverage data for client and server processes as you run the client application.

To collect data simultaneously from a client computer and a remote computer, install DevPartner on the client and DevPartner and the optional DevPartner Server license on the remote machine. See *Installing DevPartner* (DPS Install.pdf) and the *Distributed Licensing Management License Installation Guide* (LicInst4.pdf) for more information.

In this way, you can collect data for a distributed application as it is actually deployed.

You can view session files that are created by DevPartner on the same system as your installation of DevPartner, by simply opening them in DevPartner.

If you are running the server-side program from a client application running under DevPartner, and there are:

◇ DCOM-based calls between methods in different processes, or

◇ HTTP requests between Internet Explorer as the client and IIS as the server

DevPartner automatically creates a correlated session file on the client machine. The correlated session file contains the coverage data for both the client and server portions of your application. The correlated session file appears in Visual Studio, just as any other session file does.

Collecting Coverage Data from Remote Systems

You can use DevPartner to enable automatic coverage data collection for any of your application components running on a remote system. For example, you can enable DevPartner to monitor the portion of your application running on a server system.

To enable data collection on remote systems

1 Start Visual Studio on the remote system.

2 Make sure that the data collection properties are appropriate for your application. In the Visual Studio Solution Explorer, select the relevant projects and review the DevPartner properties.

3 DevPartner restarts server processes, such as IIS, after you change options. This is necessary for changes to take effect.

4 Specify instrumentation if you are analyzing an unmanaged (native code) C++ application, or a managed code application that calls native C++ components.

◇ Select **DevPartner > Native C/C++ Instrumentation** to instrument your unmanaged code.

◇ Select **DevPartner > Native C/C++ Instrumentation Manager** to open the Instrumentation Manager dialog box to make additional adjustments to instrumentation of your unmanaged code.

5 Choose **DevPartner > Start with Coverage Analysis**, or click the **Start with Coverage Analysis** button on the DevPartner toolbar to begin your analysis session.

◇　DevPartner cleans and rebuilds your unmanaged code with instrumentation, then starts your application.

　　　　◇　DevPartner instruments managed code applications as they execute.

6　When you are finished collecting data, quit your application.

Controlling Data Collection

DevPartner gives you three ways to control when coverage data is collected during the use of your application:

◆　You can use session control icons on the DevPartner **Session Controls** toolbar to interactively control data collection as your program runs.

◆　You can use the session control API to control data collection in your program.

◆　You can use the session control file to assign session control actions to specific methods in your application modules.

For information on using session controls, see the *Coverage Analysis* section of the DevPartner Studio online help in the Visual Studio .NET Combined Collection.

Viewing Your Results

This section provides a brief description of the DevPartner coverage analysis windows that you can use to work with session data. See the *Coverage Analysis* section of the DevPartner Studio online help for detailed descriptions of how to control the presentation of data in the windows.

Session Window

When you choose to view a session, DevPartner displays the Session window in Visual Studio. The Session window contains the Filter pane and the Session Data pane.

The **Filter** pane lists the instrumented images and source files used during the session. The data that appears in parentheses to the right of each name identifies the percent of lines covered and the total number of lines for each file. The Filter pane also contains filters that define logical
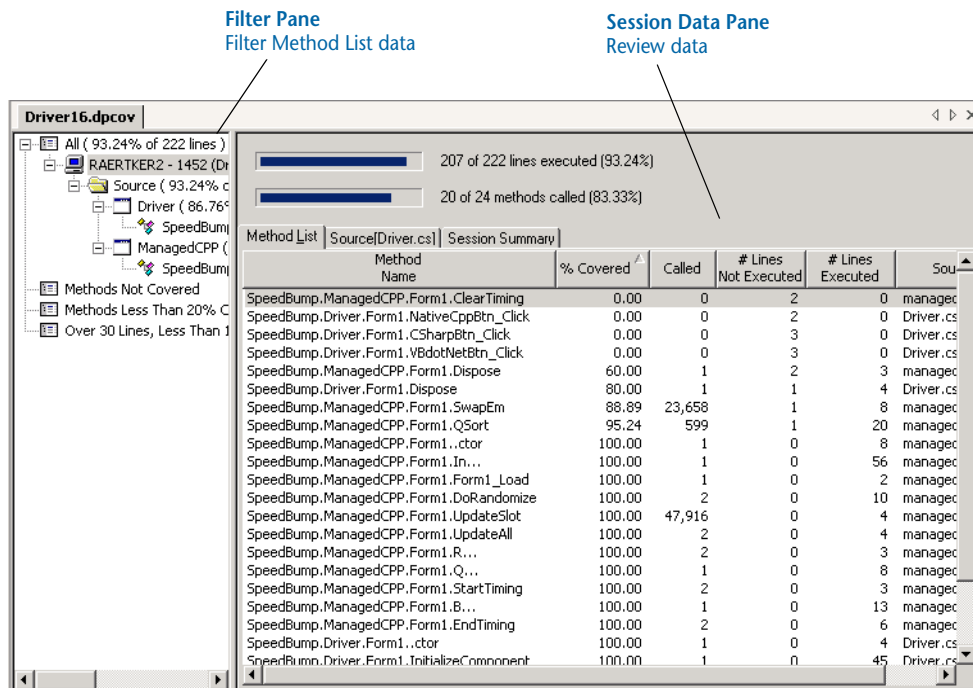
Figure 4-3. DevPartner coverage analysis Session Window

subsets of coverage data. You can click items in the Filter pane to select the subset of session data you want to view in the Method List.

The **Session Data** pane contains the Method List, Source, and Session Summary tabs.

Session Data Pane

Tip: When working in the Method List and Source tabs, right-click on the column header to show or hide data columns.

Session Data pane displays three tabs, described below, when you view data from a coverage analysis session.

◆ The **Method List** tab shows coverage data for the file, or group of files, that you select in the Filter pane. By default, the Method List tab displays all source methods.

◆ The **Source** tab displays the contents of the currently selected source file. Data columns to the left of the code provide line-by-line coverage data. For example, the Count column indicates how many times each line was executed during the session. Lines of source code are colored to indicate lines executed, lines not executed, and non-executable lines. To change the line colors:

   a   Select **DevPartner > Options**.

   b   Open the **Environment > Fonts and Colors** options page.

c Select **DevPartner Analysis** from the **Show settings for** list.

d Change the display values as necessary.

You can double-click any source file in the Filter pane to display the contents of the file in the Source tab.

◆ The **Session Summary** tab displays summary information about the session, as well as a description of the system and operating environment in which the session occurred.

Note: Method refers to the methods, functions, and procedures used by your application.

Merging Session Data

When you are testing your application using DevPartner, it is unlikely that you will execute all of your code in one session. It is important to be able to gather coverage data collected in several sessions and analyze your total coverage statistics. To accomplish this, you can merge the session files. Merging is the process of accumulating data from multiple sessions into a single file.

For example, you execute 35% of the methods in your application in Session1. If you run another session that uses some of the untested features in your application and you reach 40% coverage in Session2, you can merge those sessions to accumulate the coverage data. However, the accumulated coverage will probably not be 75% since some parts of your code were executed in both sessions. DevPartner accounts for this overlap and uses the union set of the data.

If you change your code, DevPartner tracks those changes and adjusts the coverage data accordingly. It uses merge states to distinguish between Changed, New, and Removed methods. For more information about the merge states, see the *Coverage Analysis* online help.

Files that contain merged session data are called **merge files** (.dpmrg). DevPartner can associate many merge files with a single project. They are displayed in the **DevPartner Studio** folder in Solution Explorer.

Note: You cannot merge correlated session files or Web Script session files produced from running Internet Explorer. You can merge server-side session files from Internet Information Server.

Reviewing Merge Data

DevPartner displays merge data in the Merge Data window which closely resembles the Session Data window. The Merge Data window contains the Filter and Merge Data panes. The Merge Data pane contains the

Method List, Source, Merge History, and Merge Summary tabs. The Filter pane and Source tab contain the same information they contain in Session windows.
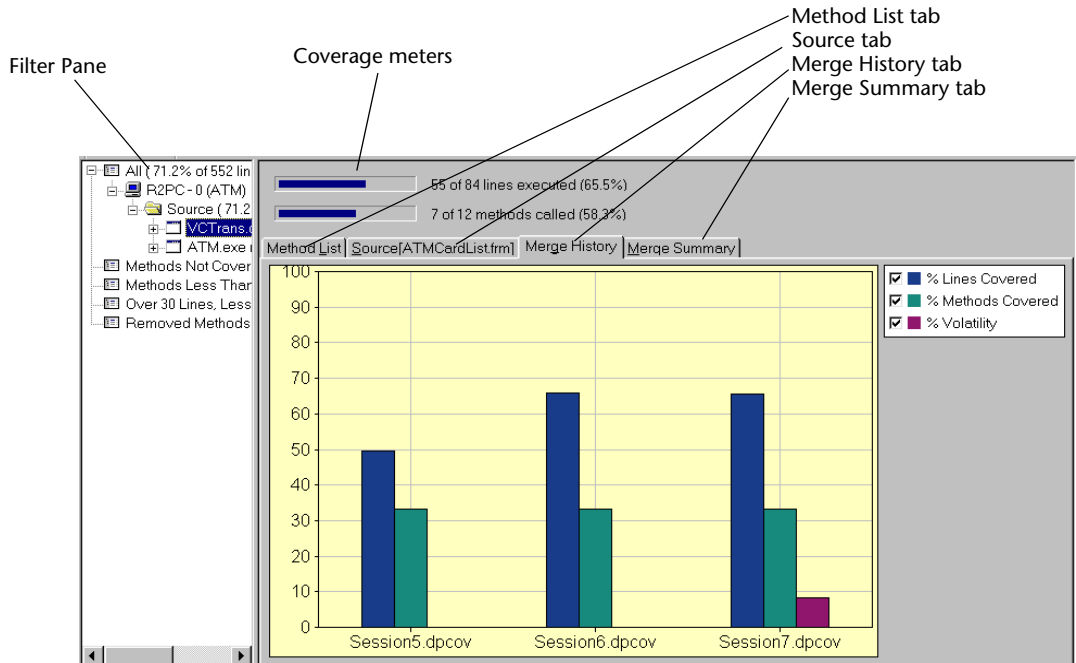
Figure 4-4. Merge Data pane

◆ The **Method List** tab uses the State column, which is not used in session files, in merge files. DevPartner uses the State column to distinguish methods that are new, changed, or removed between sessions. Refer to the *Coverage Analysis* online help for more information about method states.

◆ The **Merge History** tab displays a graphical representation of the progression of the % Lines Covered, % Methods Covered, and % Volatility values for the current merge file.

   ◇ % Lines Covered is the percentage of lines in your source code that were executed.

   ◇ % Methods Covered is the percentage of methods in your source code that were called.

   ◇ % Volatility is the percentage of methods whose source code has changed since the last merge.

◆ The **Merge Summary** tab displays summary information about the sessions and merge files that were merged into the file. It also

contains information about each of the instrumented images used during the sessions, including the % Volatility for each image.

Merge Files

You can merge session data by creating or adding data to a merge file. On the **DevPartner** menu, select **Merge Coverage Files** to create a new merge file or add data to an existing merge file. You can merge two or more session and merge files.

Merge files are useful for accumulating a subset of coverage data. You can save and merge a small set of session files to find out how much of your code was covered in a specific group of sessions. You can also save sessions, share them among team members, and accumulate the coverage data collected by everyone in a single merge file.

By default, when you close a new session that you did not merge into a merge file, DevPartner prompts you to ask if you want to merge. You can configure DevPartner not to prompt you about un-merged sessions, or to merge the coverage data automatically without prompting.

Merge Settings

When you choose a property for merging into the merge file, think about the fundamental way you want to use the merge file. To access these properties, first select the top-level solution in the Visual Studio Solution Explorer. Then, in the Visual Studio Properties window, choose a property under **Automatically Merge Session Files** in the **DevPartner Coverage**, **Memory and Performance Analysis** window.

◆ If you want to selectively accumulate coverage data in the project merge file and be asked about sessions you did not merge, use the **Ask me if I would like to merge it** setting.

◆ If you want to selectively accumulate coverage data in the project merge file and not be asked about sessions you did not merge, use **Close without prompting**.

◆ If you want to accumulate coverage data in the merge file automatically for every session, use **Merge it automatically**.

Results of Merging

When you merge session data, DevPartner:

◆ Compares percent covered values and returns the superset of the data. The percentages are not just added together, they are accumulated. For example, if you merge a session with 30% methods

covered and a session with 20% methods covered, you probably have not reached 50% coverage. There are likely parts of the code that were executed in both sessions.

◆ Calculates percent volatility values for each source and image. Percent volatility represents the % of methods that changed in your code between sessions. It demonstrates the stability of your code.

◆ Uses data from the session or merge file that ran the newest image to determine the states of your methods and images. DevPartner uses the timestamps of the images to determine which image is newest.

◆ Creates the Merge History tab and the Merge Summary tab. The Merge History tab provides a graphical representation of the progression of the % Lines Covered, % Methods Covered, and % Volatility values for the current merge file. Merge Summary tab provides the statistical data about the session.

◆ Maintains a record of all the images and methods that were loaded in any of the contributing session or merge files

◆ Maintains information about the files involved in the merge, when the merge occurred, and who performed the merge.

DevPartner treats merge data for managed code applications and unmanaged (native) C/C++ applications differently. For managed code applications, DevPartner sequentially accumulates data for code in new assemblies accessed in coverage sessions.

DevPartner collects coverage data for .NET assemblies that are loaded at execution time and for referenced assemblies whether loaded or not. When you merge coverage analysis session data collected from running a managed code application, DevPartner creates a merge file with the following characteristics:

◆ If all merged session files contain data for the same set of assemblies, DevPartner creates a merge file as described above.

◆ If you merge a session file that contains data for the same set of assemblies that exist in the merge file, plus data for additional assemblies, DevPartner adds the data for the new assemblies to the merge file.

◆ If you merge a session file that does not include data for all assemblies in the merge file, DevPartner adds data for any new assemblies, but does not alter data for existing assemblies in the merge file.

If you have removed an assembly (and its references) from the application, DevPartner places the assembly in the **Inactive Source**

filter in the **Filter** pane. DevPartner does not use inactive source in computing coverage statistics.

For unmanaged C/C++ projects, DevPartner treats code in files previously accessed, but not accessed in subsequent sessions, as inactive source and adjusts the coverage statistics accordingly.

You can create as many merge files as you wish. They are displayed in the **DevPartner Studio** folder in Solution Explorer.

Viewing Data

Controlling the Display of Data

You can control your view of coverage data by:

◆ Filtering data to show information for only the selected source files or images

◆ Sorting the source files in the Filter pane

◆ Sorting data using any of the data columns in the Method List tab

Filtering Data

The Filter pane lists the source files and images that your application used during the session and specially defined filters. You can use the Filter pane to filter data in the Method List tab and to navigate among source files and images.

Viewing Method Data

In the Filter pane, select:

◆ **All** to display data for all active images and source files.

◆ **Source** to display data for all active source methods. Double-click Source to display a tree view of the images used in your application.

◆ An application image (.EXE, .DLL, or assembly) to display data for all source methods in that image. Double-click an image to display a tree view of the source files used in that image.

◆ A source file to display data for all methods in that file. Double-click a source file to display its code in the Source tab.

◆ **Inactive Source** in a merge file to display data for all methods in images that your application loaded in earlier sessions, but did not

load during the last session. DevPartner displays the Inactive Source node only if your application did not load a previously loaded image. In managed Visual Studio applications, DevPartner places an assembly in the **Inactive Source** filter if you have removed the assembly (and its references) from the application.

◆ A filter to display data for a selected group of methods. For example, **Methods Not Covered** displays data for all the methods that you have not tested, no matter which source file or image contains them.

Viewing Source Code

Double-click any source file in the Filter pane to view its contents in the Source tab.

Displaying Source Code for a Selected Method

Complete the following steps to display the source code for a method in the Method List.

1 Click the **Method List** tab to make it active.

2 Select a method name.

3 Right-click, and on the context menu, select **Go to Method Source**.

Complete the following steps to display a specific method on the Source tab.

1 Click the **Source** tab to make it active.

2 Select the name of the method you want to view on the Coverage Session toolbar.

SpeedBump.Driver.Form1..ctor

Select Source Method

Showing and Hiding Data Columns

Complete the following steps to show or hide data columns on the Source or Method List tab.

1 Click the **Method List** or **Source** tab to make it active.

2 Right-click in a column header.

3 Choose the check boxes next to the names of the columns you want to show. Clear the check boxes next to the names of the columns you want to hide.

4 Click **OK** at the bottom of the list to update the current tab.

Sorting the Filter Pane

You can sort the source file nodes in the Filter pane by percent lines covered or name. Complete the following steps to change the sorting in the Filter pane.

1   Right-click on the **Filter** pane.

2   In the context menu, choose one of the following sort methods:
   ◇   **Sort by % Lines Covered** to sort by the percent of lines executed in the source file
   ◇   **Sort by Name** to sort alphabetically by the name of the source file
   ◇   **Sort by Unexecuted Lines** to sort by the number of lines not covered in the source file

3   After you choose a sort method, the nodes in the Filter pane collapse. When you expand them again, the source file nodes are sorted by the sort method you selected.

Creating a New Filter

Filters display a set of methods that match a specified set of criteria. After you create a filter, you can change the filter criteria using Modify Filter.

Complete the following steps to create a new filter.

1   Right-click in the **Filter** pane and choose **Create Filter**.

2   Choose the type of filter. You can filter by % Covered, total number of lines, or a combination of both.

   If you choose % Covered, specify **Minimum** and **Maximum** values.

   If you choose total number of lines, click **Between**, **Less than**, or **Greater than**, and specify the appropriate values.

3   Click **OK** to close **Create Filter** and update the Filter pane.

To modify or delete a user-defined filter, choose the corresponding option from the right-click menu in the Filter pane.

Sorting Data in the Method List

In the Method List tab, you can click a column header to sort the data by that column. Click the column header again to reverse the sort order.

Changing the Precision

All DevPartner data columns can be displayed with one to four digits to the right of the decimal point. Complete the following steps to change the decimal precision.

1 Choose **DevPartner > Options**.

2 Under the **Analysis** sub-folder, choose **Display**.

3 Use the **Precision** list to select the number of digits you want to the right of the decimal point.

Coverage Analysis for Real World Application Development

The most obvious use for code coverage analysis is in quality assurance testing. Software testers need to ensure that they have tested all parts of an application before release. Similarly, developers can use coverage analysis to ensure they have verified code integrity before significant milestones.

Merging is a powerful feature for gauging code coverage. DevPartner lets you create as many merge files as you need. For example, if your development team performs regular unit testing as application components are coded, you can merge coverage session files for each component to determine the completeness of testing before significant development milestones. Quality assurance teams can use merged coverage data to analyze the breadth of software test suites. Later they can create merge files to accumulate coverage data for scheduled quality assurance testing cycles for beta and release candidates.

*Tip: To quickly determine the last session file you merged, examine the Merge History on the merge file **Merge Summary** tab.*

If your Visual Studio application consists entirely of managed code, a merge file includes session data from all files or assemblies, including unused but referenced assemblies, that were accessed in any of the merged coverage session files. However, if your application includes native (unmanaged) C/C++ files as well, the merge file shows the native code files as **Inactive Source** if the native parts of the application were not accessed in the last session file that was added to the merge file. Thus, in order to obtain a complete coverage picture for an application that includes both native and managed code projects, make sure you run the native code portions of the application in the final coverage session you add to the merge file.

Code Coverage in the Development Life-Cycle

◆ Unit testing

As developers complete initial coding of application components, run unit tests under coverage analysis to insure the components function according to specification. Merge the results for a picture of application-wide unit testing.

If you use the DevPartner Error Detection feature as part of the development or testing process, you can accumulate coverage data at the same time you check for errors. DevPartner indicates which merged sessions were run with error detection in the Merge History section of the **Merge Summary** tab.

You cannot run performance analysis and coverage analysis simultaneously. However, when you fix a performance problem, you always verify the improvement by rerunning the test. It is a simple matter to run the test again to determine how much of your application you have tested for performance issues.

◆ Test suite development

Quality assurance teams often develop suites of tests, both manual and automated, to guarantee application functionality. Use coverage analysis to ensure that the test suites cover all significant aspects of the application. As test suites grow, you are not likely to run them all in a single session. Merge session data for tests or test suites to confirm the breadth of test coverage. You can also integrate coverage analysis into your automated regression tests.

If you make coverage analysis a regular part of your testing effort, you can set coverage percentage goals for various stages of the quality assurance process. Remember, the more code you test before you ship, the less likely you are to ship defective software.

◆ Milestone testing

Quality assurance teams typically run batteries of tests on the application before any significant milestone, such as code completion, internal or external releases. Merge the results of test sessions to ensure complete testing of the application before any release.

◆ Automated testing

You can use DevPartner to collect coverage analysis data automatically. You can also integrate unattended data collection into an automated build process using a batch file. You can do this for unmanaged Visual Basic or Visual C++ applications and for managed Visual Studio applications. Search for "automated data collection" in the Visual Studio .NET Combined Collection online help.

Running a Program from the Command Line

In addition to running your program within Visual Studio, you can use command line executables to collect coverage profiling information without launching Visual Studio.

For Visual C++ and Visual Basic applications, running your program from the command line is the only way to collect coverage data.

The following resources provide details about running your program from the command line:

◆ Refer to the DevPartner Coverage Analysis online help in Visual C++ 6.0 and Visual Basic 6.0.

◆ Refer to the section *Command Line and Configuration File Usage* in the DevPartner Studio online help within the Visual Studio combined help collection in Visual Studio 2003 or Visual Studio 2005.

Analyzing Coverage in Visual C++

You can use DevPartner to instrument your Visual C++ applications. Follow this workflow to analyze them:

◆ From the **DevPartner** menu in Visual C++, select one of the following commands to instrument your application:

   ◇ **Build > Coverage** to instrument only the new or changed parts of the active project.

   ◇ **Rebuild All > Coverage** to re-instrument the entire project.

   ◇ **Batch Build > Coverage...** to instrument or re-instrument selected files and their dependencies.

◆ Use `DPAnalysis.exe` to run your application. You can either:
   ◇ Use `DPAnalysis.exe` to run your application directly from the command line.
   ◇ Use `DPAnalysis.exe` to execute an XML configuration file to run your application.

◆ To analyze your results, use Visual Studio 2003 or Visual Studio 2005 to open the session file.

For more information, refer to the DevPartner Coverage Analysis online help in Visual C++ 6.0 and the section *Command Line and Configuration File Usage* in the DevPartner Studio online help within the Visual Studio combined help collection in Visual Studio 2003 or Visual Studio 2005.

Analyzing Coverage in Visual Basic

You can use DevPartner to instrument your Visual Basic applications. Follow this workflow to analyze them:

◆ On the Visual Basic **DevPartner** menu, select **Make with Coverage Analysis** to instrument the active project without running it

◆ Use `DPAnalysis.exe` to run your application. You can either:
   ◇ Use `DPAnalysis.exe` to run your application directly from the command line.
   ◇ Use `DPAnalysis.exe` to execute an XML configuration file to run your application.

◆ To analyze your results, use Visual Studio 2003 or Visual Studio 2005 to open the session file.

For more information, refer to the DevPartner Coverage Analysis online help in Visual Basic 6.0 and the section *Command Line and Configuration File Usage* in the DevPartner Studio online help within the Visual Studio combined help collection in Visual Studio 2003 or Visual Studio 2005.

Chapter 5
Finding Memory Problems

- ◆ Introducing DevPartner Memory Analysis
- ◆ Memory Problems in Managed Visual Studio Applications
- ◆ DevPartner Support for Visual Studio
- ◆ Identifying Memory Problems
- ◆ Locating Memory Leaks
- ◆ Solving Scalability Problems
- ◆ Managing Memory for Better Performance
- ◆ How Memory Analysis Fits in Your Development Cycle
- ◆ Running a Program from the Command Line

Introducing DevPartner Memory Analysis

The DevPartner memory analysis feature enables you to analyze memory allocation in your managed Visual Studio application.

DevPartner memory analysis presents memory data in context, enabling you to navigate chains of object references and calling sequences of the methods in your code. This provides an in-depth view of how your program uses memory and the critical information you need to optimize memory use.

Note: DevPartner Memory Analysis analyzes managed code only, and is therefore not supported in the DevPartner for Visual C++ Bounds-Checker Suite.

When you run your application under memory analysis, DevPartner can show you the amount of memory consumed by an object or class, track the references that are holding an object in memory, and identify the

lines of source code within a method responsible for allocating the memory.

This chapter describes potential memory issues in managed Visual Studio programs and shows you how to use DevPartner memory analysis to improve the performance of your applications.

Memory Problems in Managed Visual Studio Applications

Managed Visual Studio applications benefit from a sophisticated memory management environment with garbage collection. Unlike unmanaged (native) C++, in which you must explicitly free the memory you allocate, the garbage collector frees memory once the object for which it was allocated is no longer in use, or more accurately, no longer "reachable" by the application. Largely because of this feature, many developers assume that managed languages relieve them of the headaches traditionally associated with memory management. However, memory allocation and use in managed Visual Studio programs can still be a cause of performance bottlenecks and resource depletion.

Does your application exhibit any of these symptoms?

◆ Slows down over time

◆ Runs slowly, or slows down noticeably when you perform certain operations

◆ Performs poorly under load conditions

◆ Performs poorly when other applications are running

Any of these symptoms may cause you to suspect that your application has a performance problem. But how do you know if the problem is memory-related? A given number of an application's classes must be loaded before the program can execute a particular function. Is your program tying up memory resources by immediately loading classes that will not be needed unless a particular task is performed? How many instances of a particular class does your application create? How many do you really need? Similarly, every program must create and allocate objects in order to do anything useful. Object allocation always incurs memory costs. How do you know if your program is allocating too many objects, or allocating them efficiently? Are the objects your program allocates being cleared by the garbage collector? Are they being collected when you expect them to, or are they remaining in memory long after their usefulness has passed?

How Memory Analysis Helps You

The DevPartner memory analysis feature provides a comprehensive view of the way your managed application uses memory. DevPartner provides three different types of memory analysis, designed to help you isolate different kinds of memory-related problems. Regardless of which type of analysis you use, all include the following features:

◆ **Real-time graph** — DevPartner presents a **live** view of your application's memory use as it runs. This appears in the **Session Control Window**. You can see how much memory is being used by your application code (profiled code), system and other application code (excluded code), and how memory consumption compares to the memory reserved for the managed heap.

◆ **Dynamic list of classes** — DevPartner updates the list of profiled classes in real time, showing you the number of objects allocated and number of bytes used by each class, as your application runs.

◆ **Detailed heap views** — You can capture a detailed view of the managed heap at any time during program execution. DevPartner saves this data in a session file that you can then use to analyze memory problems in depth. DevPartner provides multiple ways to drill down into the session data, so you can see how your application uses memory and ultimately identify the methods or lines of code responsible for the most memory use.

DevPartner Support for Visual Studio

This section describes how the DevPartner memory analysis features integrate into Visual Studio.

DevPartner IDE Integration in Visual Studio

DevPartner is fully integrated into Visual Studio. Full integration enables you analyze the way your application uses memory on a regular basis, as you develop it, without leaving the development environment. To *-collect memory data, click **Start Without Debugging with Memory Analysis** on the DevPartner toolbar.

When the analysis session completes, DevPartner displays the data in the IDE. DevPartner saves the performance data in a memory analysis session file, with a .dpmem extension. Session files are automatically added to the **DevPartner Studio** folder that you can view in the Solution Explorer for the active solution. To review an existing memory analysis session file, double-click the file in Solution Explorer.

DevPartner Toolbar and Menu Integration

DevPartner adds commands related to memory analysis to several Visual Studio menus. In addition, DevPartner provides toolbars that include shortcuts to basic analysis functions.

◆ **DevPartner toolbar**

◇ ▣ Starts memory analysis data collection. Menu equivalent: **Start Without Debugging with Memory Analysis** on the **DevPartner** menu.

◆ **Memory Analysis Session Controls**

The session controls for memory analysis are located in the **Session Control Window**. The **Session Control Window** appears when you start your application under memory analysis. Use the **Session Control Window** to interactively control the memory analysis session. When you notice something interesting in the real-time graph, or in the dynamic list of classes, you can manipulate data collection or take an immediate snapshot of the managed heap.

◇ ▦ **Starts** and **stops** tracking object allocations (Leak analysis only)

◇ ▢ Forces a **garbage collection**

◇ ✕ **Clears** data collected to the point at which the Clear action executes (Temporary objects analysis only)

◇ ▐▌ **Pauses** and restarts the real-time graph display

◇ Process list indicates the process for which DevPartner is collecting data

◇ View... buttons take a snapshot of the managed heap.

◆ **DevPartner > Options**

◇ The **DevPartner > Analysis** folder in the **Options** dialog box accesses the **Analysis** options pages to configure memory analysis data collection.

See the Memory Analysis section of the DevPartner online help in the Visual Studio Combined Collection for more information. The online help provides details about memory analysis commands that DevPartner adds to the Visual Studio menus and context menus, and how existing Visual Studio menu commands relate to DevPartner memory analysis.

Solution Explorer Integration

When you run a memory analysis session from Visual Studio, DevPartner adds the resulting session file or files to the **DevPartner Studio** folder in Solution Explorer. You can save or discard session files when you close the solution, or set file management options in DevPartner or Visual Studio.

From Solution Explorer you can open a session file by double-clicking, use the file or folder context menu options to add or remove files from the solution, view properties, and cut or copy the file specification to the clipboard.
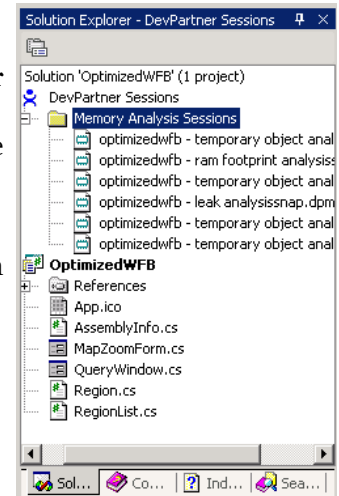
Session File Integration

When your application stops, DevPartner displays the results of the memory analysis sessions in a **Session** window in Visual Studio. From the Session window, you can analyze results within the development environment. You can drill down into the data to examine object references or to examine the call relationships of the methods that allocated the objects, jump to the source code for a particular method, and open the source code for any method for editing in Visual Studio.

Identifying Memory Problems

Consider this scenario:

When your Quality Assurance team reports the first test results for your new managed application, you were pleased to learn that it did what it was supposed to do. But in later tests, QA ran longer test cycles and reported that the longer the application ran, performance began to slow.

That is not what you wanted to hear. How do you know what part of your application to examine first? When you find the problem, how do you correct it?

To find problems in your application, run it under DevPartner. You do not have to wait until you suspect a memory problem to use DevPartner. Make testing your application's memory use with DevPartner a routine part of the development process.

DevPartner can help you quickly determine the way your application uses memory resources, revealing current or potential problem areas.

To run an application under DevPartner memory analysis:

1 Open the solution for the application in Visual Studio.

2 Review the **DevPartner Coverage, Memory and Performance Analysis** properties in the Visual Studio properties window.

3 Click **Start Without Debugging with Memory Analysis** on the DevPartner toolbar.

4 Use the **Session Control Window**, which appears after you start your application, to observe the way your program is using memory.

  ◇ The real-time graph presents a visual representation of memory use.
  ◇ The class list updates dynamically to show the classes that use the most memory as your program runs.
  ◇ Right-click the class list to switch between **Show Top 20 Classes** and **Show Top 20 Classes with Source.**

5 Use the **Session Control** buttons on the **Session Control Window** to control data collection and take snapshots of the managed heap for detailed analysis.

When you run a memory analysis session, you can choose to examine one of three important potential problem areas:

◆ Memory leaks
◆ Temporary object creation
◆ Overall RAM footprint

Table 5-1. Symptoms and Analysis Tools

Symptom	Analysis Tool
Performance degrades over time; recovers on restart. Performance improves after restarting the application, but degrades again.	Memory Leaks
Scalability problems; temporary performance degradation.	Temporary Objects Memory Leaks
Sluggish performance, does not improve after restarting the application.	RAM Footprint Temporary Objects

The symptom your application exhibits should guide your choice of the type of analysis to run first. Of course, you will eventually want to run your application under all three types of memory analysis. Even if you do not find a problem, it will enhance your understanding of how your program is using memory resources.

Collecting Server-side Memory Data

You may want to collect memory analysis data for parts of a Web or client/server application. With DevPartner, you can collect memory data for managed code in any process as you run the client application.

To collect remote process data, install DevPartner on the client and DevPartner and the optional DevPartner Server license on the remote machine. In this way, you can collect data for a distributed application as it is actually deployed. See *Installing DevPartner* (DPS Install.pdf) and the *Distributed Licensing Management License Installation Guide* (LicInst4.pdf) for more information.

Collecting Data from Multiple Processes

Web or client/server applications may run more than one process. DevPartner collects memory analysis data for managed code applications. For example, when you profile an ASP.NET application you will not collect data for the browser process (`iexplore`). However, you will collect data for managed code that runs in the `aspnet_wp` or `w3wp` processes.

When you run such an application under memory analysis, the Memory Analysis Session Control window in Visual Studio displays the server and surrogate processes in the process selection list. Use the process list to focus data collection.

Running a Memory Analysis Session

The first thing you will notice when running any memory analysis session is the real-time graph on the **Session Control Window**. The real-time graph provides a visual representation of how your application is using memory resources. Observe the pattern the graph takes as you exercise your application. Different memory problems create characteristic patterns, so the real-time graph provides the first clue to the existence and nature of a memory problem.

Tip: Pay careful attention to the shape of the real-time graph as you run your application. You can often diagnose a memory problem immediately by observing the pattern of the graph.
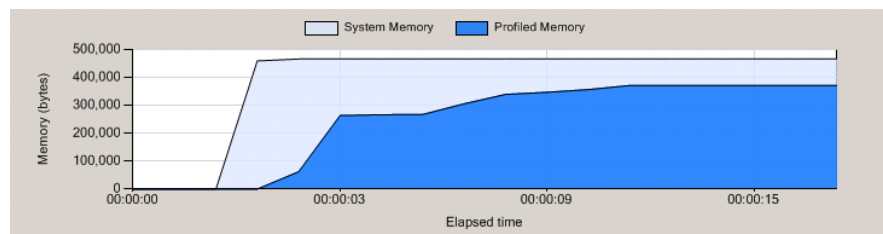

Figure 5-1. Memory Analysis **Session Control Window** Real-time Graph.

For example, if the graph shows a rising pattern that never returns to baseline, as in Figure 5-1, your application is probably leaking memory. You may suspect that the progressive slowdown of your application noticed by your QA team is consistent with a memory leak, but the real-time graph will confirm that diagnosis.

If the graph does return to baseline, but is characterized by periodic spikes in memory use, your application is creating large numbers of objects as it runs. Granted, the memory allocated is being freed, but such an application may not scale well under load.

If your application slowdown occurs in response to an increase in users or inputs, it could indicate a scalability issue. Again, the real-time graph will indicate the nature of the problem, enabling you to immediately point your diagnostic efforts in the right direction.

Even in the absence of a suggestive pattern, the real-time graph can provide important information. For example, if your application consistently consumes nearly all the memory allocated for the managed heap, and that amount is large relative to the anticipated resources of your target users' systems, you may want to reduce the overall memory footprint of your application. This chapter provides detailed information about such cases and their implications for application performance.

Locating Memory Leaks

The amount of memory consumed by your application can have a major impact on how well the application performs. The larger the amount of memory consumed (RAM footprint), the more likely it is that the application will run slowly and scale poorly. Leaked memory—the allocation of memory that is not reclaimed—can bloat your application's RAM footprint. But memory leaks do not occur in the garbage-collected managed .NET environment, do they? Granted, automatic garbage collection relieves you of the C++ programmer's responsibility to explicitly free the objects you create, so memory is not "leaked" in the classic C++ sense. But it is still possible to retain references to objects that the program does not need, and in some cases, will never use again. As long as a reference to an object exists, the referenced object is considered to be a **live object** by the garbage collector; a live object cannot be collected. This condition, like leaked memory in C++, is undesirable. Such references can be difficult to track down. That is where memory analysis helps you.

To begin, consider memory leak analysis.

Running a Memory Leak Analysis Session

To run your application under memory analysis and analyze for leaks:

1   Open the solution for your application in Visual Studio.

2   Review the **DevPartner Coverage**, **Memory and Performance Analysis** properties in the Visual Studio properties window.

3   Click **Start Without Debugging with Memory Analysis** to start your application. When the **Session Control Window** appears, select the **Memory Leaks** tab.

Tip: You can also start your application with debugging. Click the arrow to the right of the memory analysis button and select Start with Memory Analysis.

4   Exercise the program to allow any one-time initializations to complete.

5   To begin tracking memory allocations, click **Start Tracking** (🔲) in the **Memory Leaks** tab of the **Session Control Window**.

6   Exercise the program features you want to test.

You will see a spike in the real time graph, and the most memory-intensive classes will appear in the list of profiled classes. The number of tracked objects for each class appears in the **Tracked instance count** column.
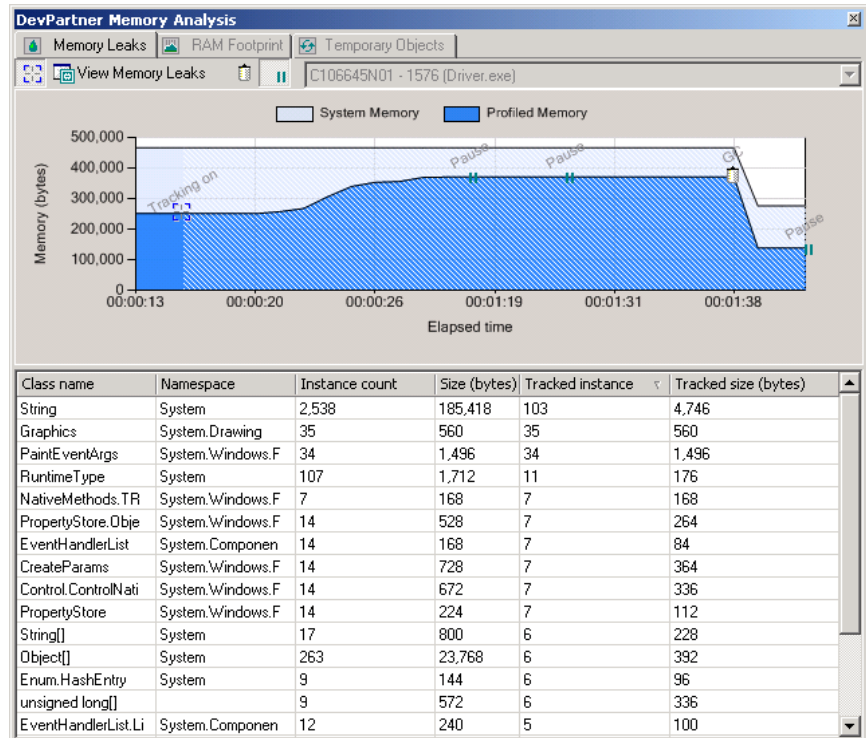
Figure 5-2. **Session Control Window** Data Display

7 You can click **View Memory Leaks** to take a snapshot of the managed heap immediately, or exercise the same section of your application again, which you would expect to clear the previously allocated objects.

8 Notice the **Tracked instance count** column in the list of profiled classes. Did the count of tracked objects decrease as you expected? Do tracked classes that you expected to be collected still appear? If you see anything suspicious, click **View Memory Leaks**.

If all the allocated objects were not freed, you should also notice that the memory display in the real-time graph did not return to the pre-exercise level.
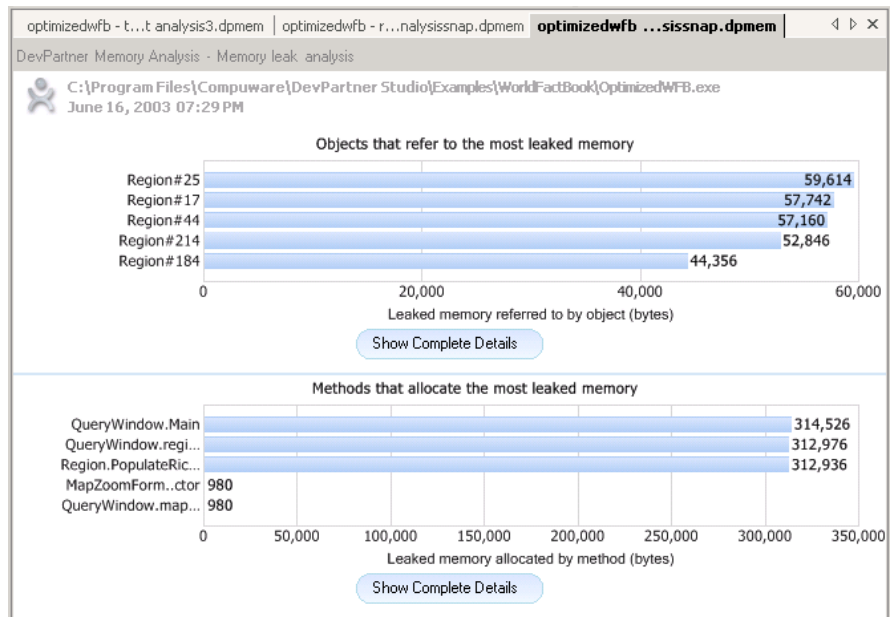
Figure 5-3. Results Summary appears when you click **View Memory Leaks**.

When you click **View Memory Leaks**, DevPartner forces a garbage collection and creates a **session file** that you will use to locate the source of the leaks. DevPartner presents the initial view of the session data in the **Memory leak analysis** page.

Understanding Memory Leak Analysis Results

DevPartner memory leak analysis defines a memory leak as any object that is allocated on the managed heap during a specified period of time, and has not been freed when you collect memory data. Memory leak analysis helps to reveal where your application holds memory that

should be freed. You can use this information to determine how to change your code so this memory will be freed.

To uncover memory leaks, run your application under the DevPartner memory leak analysis feature and exercise it in a way that should free previously allocated objects.

If memory use consistently rises and does not decrease (or does not decrease as you would expect it to) in response to garbage collection, your application is probably leaking memory.

For an example, see Figure 5-2 on page 109. The real-time graph in this figure shows a rise in memory use that did not return to baseline after garbage collection. If you look at the **Tracked instance count** for the classes that belong to your application, you will notice that some tracked objects are not being collected by garbage collection. The number of uncollected instances can be seen in the **Tracked instance count** column in the Session Control window.

Once alerted to a possible leak, use the Memory Leaks Results Summary (session file) that DevPartner creates to locate the source of the leak so you can fix it. The memory leak analysis results summary gives you the following ways to drill down into your data:

◆ Objects that Refer to the Most Leaked Memory
◆ Methods with the Most Leaked Memory

Each chart shows the top five objects or methods that are associated with leaked memory. To see more information about the top five objects or methods, click **Show Complete Details** for that chart.

The starting point you choose will depend on the problem you want to solve and your preferred approach to the problem. For example:

◆ If you notice that a limited set of specific objects are being leaked, you can use the **Objects that refer to the most leaked memory** graph to quickly see which objects hold references to the leaked objects.

◆ If you are familiar with the source code for the allocating method and can tell by examining the source code whether the leaked object should have been cleared, you may want to start with the **Methods that allocate the most leaked memory** chart.

From both the objects and methods charts, you can quickly switch to a view that shows another aspect of your data.

When viewing complete details for **Objects that refer to the most leaked memory**, you can select these views:

◆ Object Reference Graph

- ◆ Allocation Trace Graph
- ◆ Source

When viewing complete details for **Methods that allocate to the most leaked memory**, you can select these views:

- ◆ Call Graph
- ◆ Source

The following example uses **Objects that refer to the most leaked memory** as a starting point.

Objects that Refer to the Most Leaked Memory

This example shows a case where leaked memory is caused by a limited set of objects. Other possible approaches are also presented.

The garbage collector cannot clear an object as long as there is at least one existing reference to that object. When your application runs, it creates objects. Some objects are needed for as long as the program runs. These are permanent, or long-lived objects. However, most objects should become eligible for garbage collection once they are no longer referenced by another object.
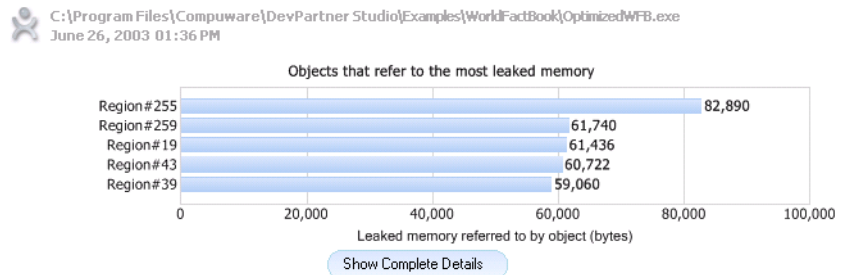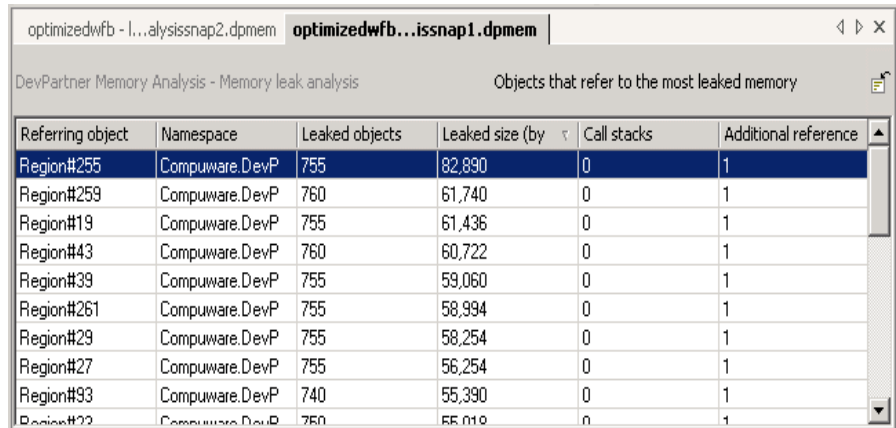
Figure 5-4. Objects that Refer to the Most Leaked Memory.

The chart in Figure 5-4 shows the top five objects that hold references to the most leaked memory. These objects prevent the leaked objects from being freed. Referring objects that account for the biggest memory hit appear at the top of the chart. The data indicates a particular set of objects is responsible for the leaked memory. Use this chart as the starting point to drill into session data and locate the source of the leaks.

Clicking **Show Complete Details** (below the bar graph; see Figure 5-4) opens a detailed display for objects that refer to leaked memory.

The top panel of this display lists all of the objects that refer to leaked memory, as displayed in the chart in Figure 5-5. This list includes the top

five objects displayed in the original bar graph and other objects that refer to smaller amounts of leaked memory.

Figure 5-5. List of objects that refer to the most leaked memory.

The default is to sort the objects by the **Leaked size** (total size of the leaked objects referred to by the selected object) column. You can also sort the list by any of the other columns, to help you see patterns in the data. If you right-click an item in the list and choose **View leaked objects referenced by this object**, you will see the objects that were actually leaked.
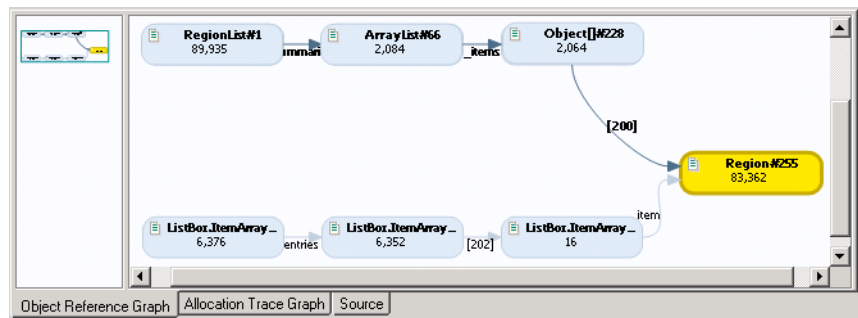
Figure 5-6. The Object Reference Graph shows why an object is still in memory.

Select a referring object that you want to examine. It is important to be able to quickly understand the sequence of references that keep these objects in memory. Click the **Object Reference Graph** tab to view the reference graph. The Ob**ject Reference Graph** shows why the selected object has not been cleared by garbage collection. It shows the chain of objects between the selected object and the garbage collection root(s) that are keeping it alive.

Scroll down the list of objects to evaluate the other objects. Some object reference graphs will be quite simple, while others may be quite complex. You may find evidence that indicates conditions such as many references to small objects or a few references to large objects. The goal is to use this graph to determine the point in the chain of referring objects where it is most efficient to eliminate the leak.

The chain of referring objects shown in the **Object Reference Graph** can range greatly in complexity. In many cases, there are multiple referrers and the graph can become very complex. Drag the rectangle or click on a node in the overview pane to change the nodes displayed in the detail pane. If you are presented with a complex graph, you can simplify the view by right-clicking a node and selecting **Show Fewer Referrers**. You can also drag nodes within the graph for easier viewing.

The labels such as `elements` on the connecting arrows represent the referring data member in the next class in the graph. Bracketed numbers identify arrays. If you know your code very well, these can speed the process of zeroing in on potential problem areas.

You can also right-click a node and select **Edit Source** to open the related source code within the IDE. You can also view the related source code by selecting the **Source** tab. DevPartner highlights the line in the method that allocated the object in the graph.

To increase program understanding, you can view the source for each node in the graph sequentially and see the events that led to the allocation of the memory that leaked. DevPartner offers alternate ways to view these program events. For example, the **Allocation Trace Graph** shows who called each method that allocated the selected object.

You can go directly from the object in the list to the source code. In real-world problem solving, you should drill down using whatever method suits the problem you are trying to solve or corresponds to the way you think about your code.

Alternate Methods of Solving the Problem

The preceding example focused on use of the object reference path to locate the source of a leak. There are other ways to approach this problem. For example:

◆ Look at the **Allocation Trace Graph** to determine who called the method that allocated the object. From there go to the source code.

◆ Go directly to the source code from the list of objects.

Recall too that DevPartner presents different views of your data on the **Memory leak analysis** page. The first example used **Objects with the**

Most Leaked Memory. However, depending on the complexity of the data, or on your own preferences, you could examine a problem from any of the following graphs on the **Memory leak analysis** page.

Methods that Allocate the Most Leaked Memory

This graph, which appears in the lower half of the **Memory leak analysis** page, shows the top five methods that allocated objects that were leaked. When you click **Show Complete Details**, DevPartner provides a list of all methods that leaked objects, with access to a **Call Graph** view and to the source code for the method, if available.

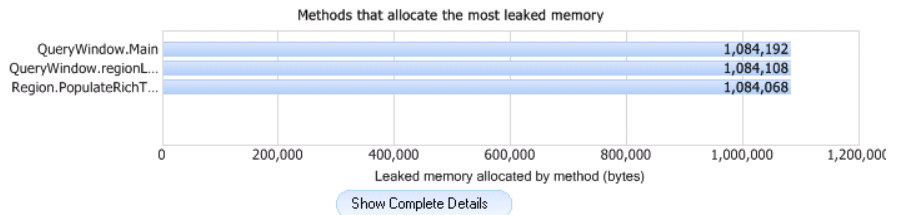Figure 5-7. Methods that Allocate the Most Leaked Memory

Select a a method in the **Method List** to see the objects allocated from the method that were leaked, or to view the source code for the method, showing the lines that allocated the leaked objects, with statistics about the number and size of objects leaked on the line.
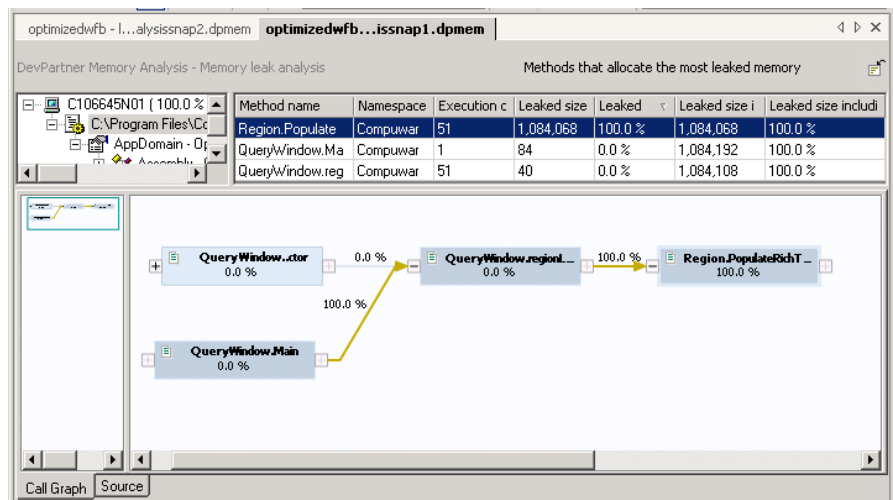

Figure 5-8. Details for Methods that allocate the most leaked memory.

For example, to drill into to the data:

◆ Right-click a method in the **Method List**.

◆ From the selected method, go to a list of the objects allocated by the method or to a **Call Graph** for the method.

◆ From an object in the Object List, view a list of referenced objects, an **Object Reference Graph**, or an **Allocation Trace Graph**.

◆ From a method, or a node in a **Call** or **Allocation Trace Graph**, view source code with object allocation data for individual lines.

◆ From a method or object in a list, or a node in a **Call**, **Object Reference**, or **Allocation Trace Graph**, or a line of source code, choose **Edit Source** to open the source to the appropriate line for editing.

Solving Scalability Problems

When performing memory analysis with DevPartner, you can use Temporary Objects analysis to diagnose and correct scalability problems.

Examples of Scalability Problems

Scalability problems surface when an application runs well until users begin to work with it more intensively. For a client-server application, this might happen when the number of users increases. For a standalone application, this might happen after numerous text manipulations or mathematical computations. These can be labeled as *scalability* problems. As the scale of the work done by the application increases, performance degrades.

A Possible Cause: Temporary Objects

One possible cause of scalability problems is the creation of too many temporary objects. In this case, object creation can become a performance bottleneck—a problem that requires correction.

The creation and use of objects is important within managed Visual Studio programs. Unfortunately, some coding techniques have the side-effect of creating many objects.

Part of the problem is the creation of objects such as those created with the String class. It takes processing cycles to create objects and later destroy these objects. If you can reduce the number of objects created, you can generally expect better performance.

Object Life Span

DevPartner tracks the objects allocated by your code and categorizes them based on how long it takes for them to be collected. There are three categories:

◆ Short-lived — collected at the first garbage collection after the object was allocated (generation 0)

◆ Medium-lived — collected at the second garbage collection after allocation (generation 1)

◆ Long-lived — survives across many (or all) garbage collections during the run of the program

Note: The Microsoft .NET garbage collector supports three generations, designated 0, 1, and 2. Objects allocated since the last run of the garbage collector are in generation 0. Objects that survive one garbage collection after allocation become generation 1 objects. Generation 1 objects that survive one or more additional garbage collections become generation 2 objects.

DevPartner combines short-lived and medium-lived object allocations in a temporary category. In other words, temporary objects are the short-lived objects and medium-lived objects that your application creates, considered as a single group.

Medium-lived objects have the greatest impact on performance, and cause the garbage collector to work harder than necessary. Individual short-lived objects have less impact on garbage collection, although there is still a performance penalty for calling the object's constructor. However, creating large numbers of short-lived objects may cause bottlenecks and memory shortages.

If you believe that your code has scalability issues, use DevPartner to monitor memory used by your code as it executes. If the real-time graph in the **Session Control Window** shows an up-and-down, wavelike pattern—which suggests that your application is creating many temporary objects—you can use DevPartner to analyze the application for temporary object creation. DevPartner categorizes the results of temporary object analysis by entry points and by methods. Regardless of which you use to drill into the data, DevPartner helps you see how much memory the temporary objects consume and identify the specific lines of code that allocate the temporary objects.

Running a Temporary Objects Analysis Session

To run your application under memory analysis and analyze for temporary objects:

1   Open the solution for your application in Visual Studio.

2   Review the **DevPartner Coverage**, **Memory and Performance Analysis** properties in the Visual Studio properties window.

3   Click **Start Without Debugging with Memory Analysis** to start your application. When the **Session Control Window** appears, select the **Temporary Objects** tab.

*Tip: You can also start your application with debugging. Click the arrow to the right of the memory analysis button and select **Start with Memory Analysis**.*

4   Exercise the program to allow any one-time initializations to complete. Exercise the features you want to test.

5   There are two ways to capture temporary object data.

    ◇ For an overall view of the way your application allocates temporary objects, click **View Temporary Objects** in the **Session Control Window** to capture a snapshot of temporary object allocations.

    ◇ To focus the analysis on a specific part of your application, click ✕ to clear the temporary object data collected as you warmed up the application. Exercise the feature of interest, then click **View Temporary Objects**.

6   When you finish data gathering, exit the application. DevPartner creates a final temporary objects snapshot.

Identifying Scalability Problems

DevPartner enables you to locate potential trouble spots and then drill down into your application's use of temporary objects to identify problems and improve the overall quality of your code.

Real-time Graph

The real-time graph provides a high level view that enables you to identify areas that may be causing problems.

If your application is creating large numbers of short and medium lived objects, you will see a peak in profiled memory in the real time graph which diminishes when the garbage collector runs. If you exercise the

feature again after garbage collection, you will see another peak, caused by creating a new group of temporary objects.
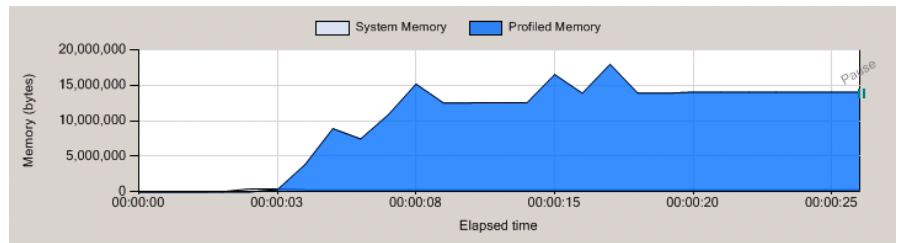
Figure 5-9. Real-time graph showing a pattern that suggests excessive temporary object creation

The classes responsible for the most objects appear in the list of profiled classes, sorted by the **Size** column. This highlights the classes whose objects consume the most temporary memory. Notice the **Instance Count** as well, which shows how many object instances were created for each class.

Figure 5-9 shows a real-time graph that suggests excessive temporary object creation. Spikes in the graph show where your application is creating lots of objects. Excessive object creation can create major performance or scalability issues in a managed code application, especially in server applications. Even if scalability is not an issue, methods that allocate many short-lived objects often indicate easy-to-fix performance problems.

Viewing Temporary Objects

Click **View Temporary Objects** to collect data at a specific point in your application. DevPartner displays a **Temporary object analysis** page that categorizes the data by entry points and by methods that create the most temporary objects.

An **entry point** is a profiled method that is called by excluded (that is, system or third-party) code. When your application runs, monitoring begins with the first call to a profiled, or user-code, method. (User-code methods are methods in your application source code.) This is called an entry point. All calls made to other user-code methods from that point are considered to be part of the entry point.

Methods that are called only by other user-code methods are not entry points. However, such methods could be responsible for large amounts of temporary memory use. The second chart on the Results Summary highlights methods that allocate lots of temporary memory, but are not necessarily entry point methods. Thus, if a child method called by an

entry point is the major memory allocator in your application, you can locate that method in **Methods that use the most memory** without having to follow the **Call Graph** for the entry point method that called it.

From the Results Summary view you can drill into the data in order to understand how much memory the objects allocated by these methods are consuming, and to identify the lines of code that are creating the short- and medium-lived objects.

Analyzing Temporary Object Data

Clicking **Show Complete Details** below either chart opens a detailed view of all the entry points, or all the methods in your application that allocated temporary memory. In addition to the complete list of methods, the view includes a **Call Graph** and a **Source** tab.

The available data columns in the Method List provide more extensive data about your application's methods than those in the list of profiled classes on the **Session Control Window**. See the online help for complete details.

Call Graph

Click on an entry point in the entry points list or a method in the method list to view a **Call Graph** for the method. The **Call Graph** shows the selected method and its child methods, and highlights the **critical path**. The critical path is the sequence of child method calls that resulted in the largest cumulative memory allocation for the selected method.

Methods appear as nodes in the Call Graph. Each node can display data about memory allocated by the method. In addition, the links between nodes can display data about memory allocated by that branch of the graph. The data is expressed as percentages of memory allocated.

◆ **Nodes -** The percentage of memory allocated by the method that is attributable to the body of the method itself.

◆ **Links -** The percentage of memory allocated by the method that is attributable to child methods executed in that branch.

In this way DevPartner shows you not only which methods are responsible for the temporary objects your application creates, but exactly where in the paths of execution the allocations occur.
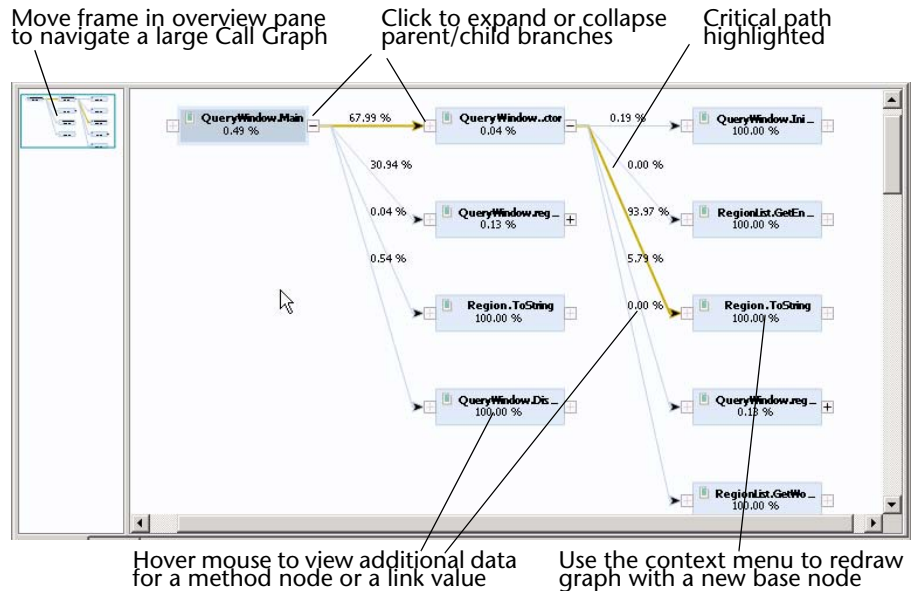
Move frame in overview pane to navigate a large Call Graph

Click to expand or collapse parent/child branches

Critical path highlighted

Hover mouse to view additional data for a method node or a link value

Use the context menu to redraw graph with a new base node

Figure 5-10. A Call graph for an entry point method shows the critical path.

Right-click on any node in the Call Graph to:

◆ Redraw the Call graph for the selected node
◆ View source code for the selected node
◆ Edit source code for the selected node

Source View

When you view the source code for an entry point or method in the **Entry points that allocate the most memory** or the **Methods that use the most memory** method lists, DevPartner opens the Source tab view to the selected method. In addition to the source code, DevPartner provides detailed information about the memory allocated by individual lines in your application code, including how often the line executed, the number of short-, medium-, and long-lived objects, including or excluding child objects, allocated on the line, and the accumulated sizes (memory load) of these objects. See the online help for a complete list.

Interpreting Results to Fix Scalability Problems

The following suggest some of the possible ways to interpret memory analysis results to fix memory-related scalability problems.

◆ Look at the **Temporary Objects analysis** page to determine if an entry point or non-entry point method consumes the most temporary memory.

- If the largest consumer is an entry point, drill down using the **Entry points that allocate the most memory** view to determine which methods in the entry point's execution path require the most temporary space and should be modified or called less often.

- If the largest consumer is an non-entry point method, drill down using the **Methods that use the most memory** view to determine which parts of your code to modify.

- Compare the number of short- and medium-lived objects, as well as the amount of temporary space they consume. Use this information to determine which parts of your code to modify.

- If both short-lived and medium-lived objects consume similar amounts of temporary space, you can run a performance analysis to find out how much time the constructor uses to create the temporary object.

- Use the **Call Graph** to understand the relations between methods that allocate temporary memory. Examine characteristics of different methods: percentages of memory consumed; actual bytes used; numbers of temporary objects created. Use this information to identify which method to modify.

- Use the **Source** tab to identify specific lines in your code that allocate temporary objects. Examine the kind and sizes of objects created, and how often the line is executed. Use this information to identify more efficient ways to use objects.

Managing Memory for Better Performance

Some managed code applications can consume hundreds of megabytes of RAM while they are running. We have examined some specific memory problems in this chapter: memory leaks, which can cause your application to consume more and more memory as it runs until it eventually exhausts the heap; and periodic spikes in memory use caused by excessive temporary object creation, which can lead to scalability issues. These problems impact your application's memory use in negative ways. These problems also contribute to your application's memory footprint. Your application may be well-behaved with respect to these errors—but it may still feel slow, especially when run on the kinds of hardware and in the kinds of software environments that your target users are likely to have.

One probable cause of sluggish performance is that your application may be using excessive amounts of memory as it runs. What is excessive? That

depends on the environment—hardware and software—in which your application is used. You may have a pretty good idea of what that environment is, but you cannot know for certain that your target users will not try to run several other applications at the same time as they run yours. Nor can you force hardware upgrades on your users every time you release a new version of your application. All of this makes a strong argument for keeping your application's memory footprint small.

Let's be more specific: We are talking about RAM footprint, not just overall memory use. The worst thing you can do to ruin application performance—and your end-users' perception of your application—is to force it to rely on the operating system's virtual memory system. Paging managed objects into virtual memory is a sure way to ruin application performance.

Tip: Test frequently during development to make sure your application continues to exhibit an acceptable footprint.

What can you do to optimize your application's use of RAM resources? DevPartner provides RAM Footprint analysis as part of its memory analysis capability. Run RAM Footprint analysis regularly as you develop your application. The way your application uses RAM resources is most likely a result of application design and architecture. It is much easier to re-design a feature early in the development process than to wait until the application is ready for beta release.

Measuring RAM Footprint

DevPartner helps you focus your tuning efforts on the areas that will have the greatest impact on RAM consumption.When you run your application under RAM Footprint analysis, DevPartner enables you to:

◆ View the real-time graph of your application's RAM consumption, and view the real-time list of profiled classes associated with the most bytes of memory.

◆ Take snapshots of the managed heap, which you use to examine the objects responsible for the most memory use.

To run your application under RAM Footprint analysis:

1 Start your application under a DevPartner memory analysis session.

When the **Session Control Window** appears, click the **RAM Footprint** tab.

2 Exercise the application to allow classes to load and one-time initializations to run.

As you run the application, observe the real-time graph. Be alert for patterns that may indicate a specific problem, such as a leak or excessive spiking due to temporary object creation.

Notice the amount of RAM consumed. The y-axis of the real-time graph indicates the amount of heap memory used by your application (Profiled Memory). Notice also the amount of RAM being used in relation to the memory allocated for the managed heap (System Memory).

3   Warm up the application and get it into a steady state. (The application is probably idle at this point.)

4   Force a garbage collection.

5   Click **View RAM Footprint** to generate a RAM Footprint Results Summary (session file) reflecting the current state of the managed heap.

Understanding Footprint Data

Use the RAM Footprint analysis page to gain an in-depth understanding of how your application is using memory. The RAM Footprint Results Summary gives you the following ways to examine and drill down into your data:

◆   Object Distribution
◆   Objects that refer to the most allocated memory
◆   Methods that allocate the most memory

Which you use first will depend on the data presented, and to some extent, on the way you tend to think about your application.

Object Distribution

DevPartner presents the distribution of objects in memory as a pie chart so you can immediately see the proportion of memory used by your application (Profiled objects) relative to that used by system code (System objects).

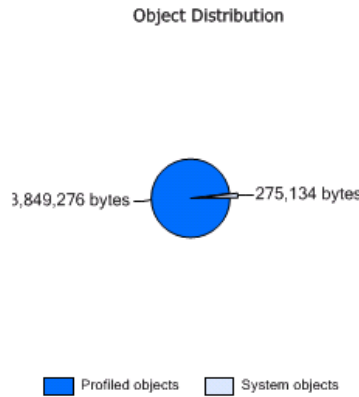Interpreting the Object Distribution chart:

Object Distribution

3,849,276 bytes ———   —275,134 bytes

▢ Profiled objects   ▢ System objects

Figure 5-11. Object Distribution Chart

◆ If your application (profiled code) is the largest wedge in the pie, and memory use is moderate to high relative to expected resources in the target deployment environment, you should determine which parts of your application allocate the most memory. To do this, use the **Objects that refer to the most allocated memory** or the **Methods that allocate the most memory** chart to drill down into the data. Ultimately, you want to locate in source code those parts of the application that you can change or restructure to use less memory.

◆ If the Profiled Objects part of the pie chart is small, your application is not the main allocator of memory. This is a good thing. But if the application still seems sluggish or if overall memory use is high, you may want to investigate how your application is using unmanaged code or system resources. Unmanaged code can pin objects in memory. Visual Studio applications often spend a great deal of time in the .NET Framework; you may find that you can call Framework methods more efficiently, or less often.

You can drill into the RAM footprint data by using either of two analysis paths.

◆ Objects that refer to the most allocated memory

◆ Methods that allocate the most memory

Objects that refer to the most allocated memory

Objects that refer to the most allocated memory shows the objects that held references to live objects at the time the session file was generated. The size displayed is the total of all objects referenced from this instance.

◆ Click **Show Complete Details** to drill into the data for these objects.

Objects that refer to the most allocated memory enables you to focus on instances of objects responsible for the largest amounts of memory. Organizing the data by instances of objects that hold references to allocated memory highlights **large objects**, that is, the objects for which

a maximum amount of memory would be reclaimed if the object could be garbage collected.

While an individual object might be small, it becomes much larger, i.e., a large object, when you include the memory consumed by the objects to which it refers. When the garbage collector runs, it cannot collect objects that are referenced by other objects. Thus an object that refers to many other objects may account for a considerable amount of memory. If you can collect such an object, you can also collect any other objects to which it holds a unique reference. Such large objects are obvious targets when you are trying to reduce an application's RAM footprint.

The **Objects that refer to the most allocated memory** view includes a table of live object instances with data about each object's impact on memory at the time the session file was created. It also includes a tabbed window in which you can view an **Object Reference Graph**, **Allocation Trace Graph**, and **Source**.

This view helps you identify the largest objects in memory. Referenced Size data includes memory attributable to all child objects for which the object is the only parent. Considered singly, objects tend to be small. However, an object with several child objects, each of which may also have child objects, plus per-object overhead for parent and child objects, is actually consuming a large amount of memory. In essence, DevPartner utilizes the Object Reference path to roll up the bytes associated with child objects and attributes them to the parent object. The advantage of this view is that it lets you focus on those objects that will provide the biggest benefit if you can change the way they are allocated.

Once you zero in on the objects that consumed the largest amount of memory, you may immediately see changes you could make to reduce memory consumption. However, you may want to investigate further to understand the implications of freeing or changing the way the application uses a particular object.

◆ Double-click the selected object in the instance list, or use the context menu to view the live objects referenced by the selected object.

Live Objects Referenced by *object*

The **Live objects referenced by object** view shows you all of the live objects, that is, objects still in memory, that are referenced by the selected parent object. Put another way, these are the child objects that could also be collected if the parent object could be collected.

All Objects Referenced by *object*

The **All objects referenced by object** view displays an instance list of the all of the objects referenced by an object selected in the Live Objects Referenced by object window.

Like its parent windows, the data presented in All Objects Referenced by object is organized by instances of objects that hold references to allocated memory. This view enables you to further examine the chain of references keeping objects in memory. In the All Objects Referenced by object window, you can examine the entire chain of objects referenced by any of the child objects in Live Objects Referenced by object.

You can continue to drill down from any object and view all the objects to which it holds a reference through the entire sequence of object references.

The Object Reference and Allocation Trace Graphs

All of the Object views discussed above include an **Object Reference Graph** and an **Allocation Trace Graph**.

The **Object Reference Graph** shows live objects in memory at the time the session file was created. A live object is an object on which methods can be invoked. When the garbage collector runs, it identifies the objects that have valid references. A valid reference means that an object is reachable from the application's garbage collection roots. Reachable objects are marked as live objects and cannot be collected. The **Object Reference Graph** shows these object references. The graph answers the question: Why is this object still in memory?
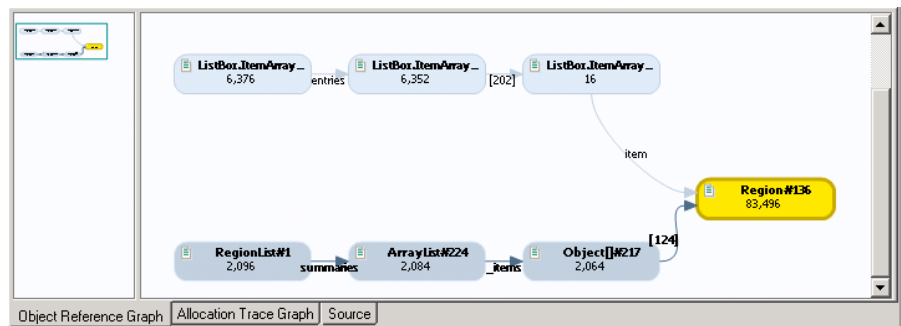

Figure 5-12. The Object Reference Graph

Objects and the memory they consume are allocated by your application's methods. It is useful to know the sequence of method calls

that allocated memory. The **Allocation Trace Graph** shows the method calls that allocated an object.
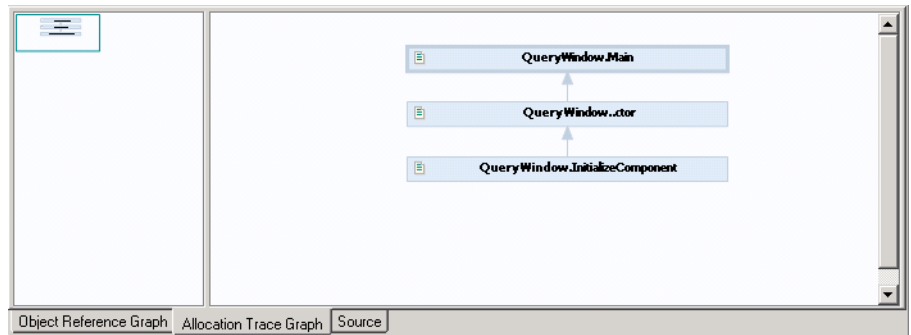

Figure 5-13. The Allocation Trace Graph

Methods that Allocate the Most Memory

The **Methods that allocate the most memory** view displays a Method List showing the source methods that allocated the most live memory for the application. This view focuses on the methods that are responsible for the greatest percentage of the total memory in the managed heap that cannot be freed by garbage collection while the application is in its current state.

A glance at the **Live size including children (%)** column indicates the percentage of memory used by the method (and its child methods) relative to total allocated memory in the managed heap at the time the session file was created. This focuses your attention on the most memory intensive methods.

In addition to a view of the source code, this view also includes a Call Graph, which shows the execution path responsible for the memory allocation. For information on using the Call Graph, See "Call Graph" on page 120.

Live Objects Allocated by This *method*

The **Live objects allocated by this *method*** view displays a list of the live object instances allocated by the method you selected in the **Methods that allocate the most memory** view. In this case the view is limited to live objects allocated by the method selected in the previous window. This enables you to drill down from those methods in your application that were the largest allocators of live memory to examine the objects

that were still alive, that is, objects that were not available for garbage collection, when the RAM footprint snapshot was taken.

Note: The list of allocated objects includes objects created by non-profiled (system) methods called by the user-code method selected in the Methods That Allocate the Most Memory view. For example, if your method uses methods in the WinForms library, objects allocated by those methods will appear in the list of allocated objects.

In order to understand how your application allocates objects, you can drill down to examine all the objects referenced by any live object allocated by the method under study. The **All objects referenced from this instance** view is identical to the view described under "All Objects Referenced by object" on page 127.

You can continue to drill down from any object and view all the objects to which it holds a reference through the entire sequence of object references.

Optimizing Memory Use

Once you understand how your application is using memory, you can begin to optimize memory use. Objects are typically the largest memory consumers, so we will take them as starting points.

Your program probably creates lots of objects as it runs. Do you simply try to reduce the number objects created? How do you know where to focus your tuning efforts?

Fortunately, DevPartner does much of the cost/benefit calculating for you. Remember that individual objects may be small, but when you consider objects with their children, some objects are much larger than others. DevPartner uses the concept of large object to alert you to those objects which, with their child objects, are large consumers of memory. Focusing your tuning efforts on these object allocations promises the most rapid route to a reduced footprint.

Be aware of medium-lived objects. These are objects allocated since a transaction started that survive to generation 1; they are collected during the second garbage collection after the transaction completes. This is the new amount of memory your transaction requires. If you could reduce the number of objects allocated, you could probably improve performance.

Look at live objects at several points as your application executes. Have you allocated objects that will not be needed for the remainder of the transaction? Are there live objects that could be shared between multiple transactions? Have you allocated any objects that the application will not need until a later time? If the answer is yes and you can change the way

your application allocates objects, you can probably reduce footprint and improve performance.

How Memory Analysis Fits in Your Development Cycle

It is not necessary to wait until you suspect that you have problem to begin testing. If you run memory analysis early and often, and know what to look for when you analyze your application, you can correct problems early, when they are both easier to track down and entail less risk to fix.

Memory problems in managed applications are often the result of larger design and architecture decisions, rather than simple coding errors. For example, one source of memory loss is an object that is not collected because of an out-dated reference to it that is not freed. This can be the result of revisions made in another part of the code. The later these problems are identified in the development cycle, the more difficult and expensive they will be to fix.

As a result, it is valuable to use memory analysis as part of a continuous testing program throughout the development cycle. You will want to use memory analysis during unit testing to get an understanding of how the individual modules handle memory. Once you identify and fix areas that need improvement, retest to verify the fix. Then, as you integrate the modules into your application, repeat your memory testing again to assure that new memory problems do not appear.

Running a Program from the Command Line

In addition to running your program within Visual Studio, you can use command-line executables to collect memory analysis information without launching Visual Studio.

Refer to the section *Command Line and Configuration File Usage* in the DevPartner Studio online help within the Visual Studio combined help collection in Visual Studio 2003 or Visual Studio 2005.

Chapter 6
Automatic Performance Analysis
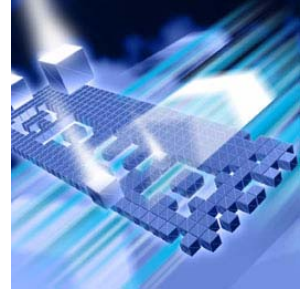
◆ Introducing DevPartner Performance Analysis

◆ DevPartner Support for Visual Studio

◆ Collecting Performance Data

◆ Controlling Data Collection

◆ Viewing Your Results

◆ Performance Analysis for Real World Application Development

◆ Running a Program from the Command Line

◆ Analyzing Performance in Visual C++

◆ Analyzing Performance in Visual Basic

Introducing DevPartner Performance Analysis

The DevPartner performance analysis feature makes it easy to pinpoint performance bottlenecks anywhere in your code, third party components, or operating system, even when source code is not available.

DevPartner allows you to gather performance details without leaving Visual Studio. You can collect performance data for a managed code application or native C/C++ application any time you run the application from the **DevPartner** menu in Visual Studio. DevPartner also lets you analyze your applications and components as they are really used — as native applications, ADO.NET components, or ASP.NET or Web applications. DevPartner can even collect data for managed code applications started outside Visual Studio.

DevPartner also supports performance profiling of legacy Visual Studio 6.0 components used by your application. See "Performance Analysis for Visual Basic and Visual C++" on page 133.

DevPartner can show detailed performance information for lines of source code (when available), methods, functions, assemblies, and executables.

How DevPartner Performance Analysis Helps You

When you run a performance analysis session, DevPartner collects performance data for images, source files, assemblies, methods, functions, and individual lines of code. DevPartner also provides end-to-endperformance profiling for distributed, component-based applications. DevPartner collects performance data for each process, including remote server session dataand COM method calls across processes and systems. The server data is transmitted back to the client and is automatically correlated with the client data into a single session file.

DevPartner gathers performance data for executable images, executable server extension code (such as an ISAPI DLL), and scripts contained in HTML files written in the scripting languages supported by Internet Explorer and Microsoft Internet Information Server (IIS). DevPartner can also profile in-process or out-of-process COM objects invoked by the scripts.

What is DevPartner Performance Analysis?

DevPartner provides essential performance analysis features for .NET applications:

- ◆ Ease of use
- ◆ Complete performance analysis
- ◆ Multi-language performance analysis
- ◆ Focused data collection
- ◆ Accurate, reproducible timing results
- ◆ Third-party component monitoring
- ◆ Export DevPartner Data

Ease of Use

DevPartner is fully integrated into Visual Studio. You can collect performance data without leaving the IDE. Finally, DevPartner presents the performance data in an easy-to-navigate window in Visual Studio. You can use these flexible features to pinpoint slow code and performance bottlenecks anywhere in your application, without leaving the development environment.

Complete Performance Analysis

DevPartner collects performance data for images, source files, methods, functions, and individual lines of code. DevPartner supports performance data collection for both managed code applications and native code C/C++ applications. You can view and sort the data in different ways to focus your analysis.

Multi-language Performance Analysis

DevPartner supports all .NET managed code languages, as well as native C/C++. DevPartner can also collect performance data for Visual Basic and Visual C++ 6.0 applications, as well as JScript and VBScript Web applications when using Internet Explorer or IIS.

Performance Analysis for Visual Basic and Visual C++

DevPartner 8.1 provides limited integration for Visual Basic 6.0 and Visual C++ 6.0. You can build your Visual Basic and Visual C++ applications with performance instrumentation in the Visual Studio 6.0 environment.

The sections "Analyzing Performance in Visual C++" on page 151 and "Analyzing Performance in Visual Basic" on page 152 provide an overview of the procedure and direct you to other sources of information.

Focused Data Collection

DevPartner Session Controls let you focus collection of performance data on any phase of your application. You can use the session controls to stop data collection, clear data collected to that point, or take a snapshot of the data currently collected and then continue collecting data.

Accurate, Reproducible Timing Results

DevPartner can distinguish between time spent in threads of your application and time spent in threads of other running applications. This enables DevPartner to generate accurate, reproducible results that are independent of outside influences.

Third-party Component Monitoring

DevPartner checks the performance of third-party controls, with or without source code. It collects data for all the ADO methods, properties, and exported functions used by your application.

Export DevPartner Data

The Export DevPartner Data feature enables you to export DevPartner performance session data files (with the `.dpprf` extension) to XML. The **Export DevPartner Data** command is available on the **File** menu.

You can analyze the data in the exported XML file using your own or third-party software. For example:

◆ Use Export DevPartner Data on a development build server or QA server where unit tests, functional tests, or regression tests are staged. Analyze the exported XML data to monitor daily progress.

◆ Use Export DevPartner Data to collect data for longer-term analysis. You can accumulate the XML data in a database or data warehouse in order to:
  ◇ Integrate the data with development and QA methodologies, tools and infrastructure
  ◇ Run custom analytics on the data
  ◇ Archive the data for historical or auditing purposes

How DevPartner Performance Analysis Fits in Your Development Cycle

In order to understand the performance characteristics of your application, you will want to use DevPartner performance analysis throughout the coding and testing phases of your development cycle. As you develop the individual components, use DevPartner to identify performance problems in the separate components. As you integrate components, test with DevPartner performance analysis again to reveal performance problems in the interactions between the components. Finally, use DevPartner performance analysis to test the entire application in different configurations and with different user scenarios to be sure that performance bottlenecks do not develop in these different situations.

DevPartner Support for Visual Studio

This section describes how the DevPartner performance analysis features integrate into Visual Studio.

DevPartner IDE Integration in Visual Studio

DevPartner is fully integrated into Visual Studio. This makes it easy to analyze the performance of an application on a regular basis, as you develop it, without leaving the development environment.

When the analysis session completes, DevPartner displays the performance data in the IDE. DevPartner saves the performance data in a performance analysis session file, with a `.dpprf` extension. Session files are automatically added to the **DevPartner Studio** folder in the active solution. To review an existing performance analysis session file, double-click the file in Solution Explorer.

DevPartner Toolbar and Menu Integration

DevPartner adds commands related to performance analysis to several Visual Studio menus. In addition, DevPartner provides toolbars that include shortcuts to basic performance analysis functions.

◆ **DevPartner toolbar**

◇ Starts performance data collection. Menu equivalent: **Start Without Debugging with Performance Analysis** on the **DevPartner** menu.

◇ Enables Native C/C++ data collection when you rebuild the solution or project. Menu equivalent: **Native C/C++ Instrumentation Manager** on the **DevPartner** menu.

◇ Opens the DevPartner options pages. Menu equivalent: **Options** on the **DevPartner** menu.

◆ **DevPartner Session Controls toolbar**

◇ ■ **Stops** data collection and takes a final data snapshot

◇ Takes a data **Snapshot**

◇ **Clears** data collected to the point at which the Clear action executes

◇ Process list focuses data collection on a single process for applications that run in multiple processes.

◆ **DevPartner Performance Session toolbar**

◇ Opens the **Choose a Basis Session for Comparison** dialog to configure a comparison of two performance sessions. Context menu equivalent: **Compare**.

◇ Opens the **Call Graph** for the selected method. Context menu equivalent: **Go to Call Graph**.

◇ **Source method selection** list locates the selected method on the **Source** tab.
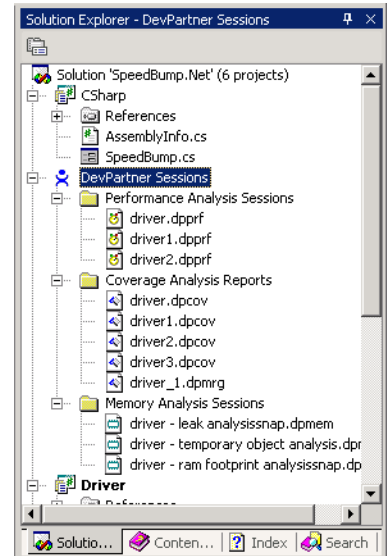
◆ **DevPartner Menu**

◇ **Start Without Debugging with Performance Analysis** starts performance data collection.

◇ **Native C/C++ Instrumentation** Activates DevPartner instrumentation.

◇ **Native C/C++ Instrumentation Manager** Opens the DevPartner Native C/C++ dialog box which enables you to set DevPartner instrumentation options for unmanaged (native) C/C++ projects before rebuilding.

◇ **Correlate > Performance Files** Combines client and server-side session files into a single correlated file.

◇ **Options** Accesses the Analysis options pages to configure performance data collection.

See the Performance Analysis online help in the Visual Studio Combined Collection for information about other commands DevPartner adds to the Visual Studio menus and context menus, and how existing Visual Studio menu commands relate to the DevPartner performance analysis feature.

Solution Explorer Integration

When you run a performance analysis session from Visual Studio, DevPartner adds the resulting session file or files to the **DevPartner Studio** sub-folder of the **DevPartner Sessions** folder in Solution Explorer.

From Solution Explorer you can open a session file by double-clicking, or use the file or folder context menu options to add or remove files from the solution, view properties, and cut or copy the file specification to the clipboard.

Session File Integration

DevPartner displays the results of the performance analysis sessions in a
Session Data pane in Visual Studio, so you can analyze results within the development environment. From the Session Data pane you can filter the view of the data, sort the data by various criteria, jump to the source code for a particular method, and access a **Call Graph** for any method to examine relationships of parent and child methods. You can also

compare sessions to examine the effect of your code changes on performance. See "Viewing Your Results" on page 141 for more information.

Collecting Performance Data

This section describes how to:

◆ Collect performance analysis data.
◆ Collect performance data for both client and server portions of a client/server application.
◆ Collect performance data for any of your application components running on a remote system.

.Running Your Program under DevPartner Performance Analysis

This section describes how to run your application with DevPartner to collect performance analysis data.

For Managed Code Applications

Many applications you will develop in Visual Studio will be managed code applications. C#, Visual Basic .NET, and managed Visual C++ applications are examples of managed code applications. In order to analyze an application as it runs, DevPartner instruments the application code for the specified type of data collection, in this instance, performance data. In the case of managed code applications, DevPartner instruments for data collection at runtime, as the application code is compiled for execution by the common language runtime.

As a result, it is easy to collect data for managed code applications.

1   Review the **DevPartner Coverage**, **Memory and Performance Analysis** settings in the properties window. Make sure the properties are appropriate for your application.

◇   To display the **Properties** window, select **View**, then choose **Properties Window**.

*Tip: You can also start your application with debugging. Click the arrow to the right of the performance analysis button and select **Start with Performance Analysis**.*

2   Click **Start Without Debugging with Performance Analysis** on the DevPartner toolbar.

3   Exercise the application. Use the session controls to focus data collection, if you wish.

4   Exit the application.

DevPartner collects performance data as the application runs, and displays the data in the main window in Visual Studio when the analysis session ends. The session data file (`.dpprf`) appears in the performance analysis sessions virtual folder in Solution Explorer. If you used the Snapshot session control, DevPartner displays (and saves) a session file for each snapshot.

For Unmanaged (Native) Visual C++ Applications

Unlike managed code, which DevPartner instruments at runtime, you must instrument unmanaged (native) C/C++ code when you compile it. DevPartner makes this as easy as rebuilding your solution or project.

Choose **Native C/C++ Instrumentation Manager** on the **DevPartner** menu. In the Instrumentation Manager, choose the type of instrumentation, in this case, performance analysis, and select the native C/C++ projects you want DevPartner to instrument. Then, rebuild the solution, or rebuild the specific projects.

Once your unmanaged C++ application or project is instrumented, click **Start with Performance Analysis** on the DevPartner toolbar.

If your Visual Studio application includes managed and unmanaged portions, DevPartner collects data for both, provided the managed and unmanaged portions are in separate files.

For Web Applications

If you develop Web Forms, XML Web services, or ASP.NET applications, you can use DevPartner to analyze application performance. If you can start your application from Visual Studio, you enable performance analysis and run the application. DevPartner collects the performance data as it would for any other managed code application.

However, there are several things you need to be aware of when analyzing Web applications.

Before you begin collecting data for analysis:

◇ *Warm up* the application by exercising it for several minutes. Be sure to include the parts of the application in which you are interested.
◇ Execute the **Clear** session control action to discard data collected to that point.
◇ Collect data.

In this way, you can eliminate data collection for the many one-time initializations that take place when you launch the application.

Collecting Server-side Performance Data

You may want to collect performance data for both client and server portions of a client/server application. With DevPartner, you can collect performance data for client and server processes as you run the client application.

To collect client-server data, install DevPartner on the client and DevPartner and the optional DevPartner Server license on the remote machine. In this way, you can collect data for a distributed application as it is actually deployed. See *Installing DevPartner* (DPS Install.pdf) and the *Distributed License Management License Installation Guide* (LicInst4.pdf) for more information.

Collecting Data from Multiple Processes

Client/server applications may run more than one process. For example, when you profile an ASP.NET application you might see the browser process (`iexplore`), the IIS process (`inetinfo`), and the ASP worker (`aspnet_wp` or `w3wp`) processes.

When you run such an application under performance analysis, the DevPartner Session Controls toolbar in Visual Studio displays the active processes in the process selection list. Use the process list to focus data collection. When you execute a Snapshot session control action, DevPartner creates a snapshot session file with data for the process selected in the process list.

Note: The process list in the DevPartner Session Control toolbar includes all active processes in the analysis session. However, if all processes run on the local machine, DevPartner launches a separate version of the Session Control toolbar for each process. These instances of the toolbar reflect only a single process. You can use the separate toolbar to execute session control actions for that process, or use the primary Session Control toolbar in Visual Studio to select any active process and execute session control actions.

Data Correlation

DevPartner not only collects data for applications that run in multiple processes, but it can automatically correlate the client and server data in a single session file. If you are running the server-side program from a client application running under DevPartner performance analysis, and there are:

◇   DCOM-based calls between methods in different processes, or

◇ HTTP requests between Internet Explorer as client and IIS as server

DevPartner automatically creates a correlated session file on the client machine that contains the performance data for both the client and server portions of your application. The correlated session file appears in Visual Studio, just as any other session file does.

When you view the correlated session data in the Call Graph window, you can navigate between called and calling methods and see the performance data associated with the call.

Collecting Performance Data from Remote Systems

You can use DevPartner to enable automatic performance profiling for any of your application components running on a remote system. For example, you can enable DevPartner to monitor the portion of your application running on a server system.

To collect data simultaneously from a client computer and a remote computer, install DevPartner on the client and DevPartner and the optional DevPartner Server license on the remote machine. See *Installing DevPartner* (DPS Install.pdf) and the *Distributed License Management License Installation Guide* (LicInst4.pdf) for more information.

To enable data collection on remote systems

1 Start Visual Studio on the remote system.

2 In the Visual Studio Solution Explorer, select the relevant projects and review the **DevPartner Coverage**, **Memory and Performance Analysis** properties.

◇ To display the **Properties** window, select **View**, then choose **Properties Window**.

3 DevPartner restarts server processes, such as IIS, after you change properties. This is necessary for changes to take effect.

4 Specify instrumentation if you are analyzing an unmanaged (native code) C++ application, or a managed code application that calls native C++ components.

◇ Select **DevPartner > Native C/C++ Instrumentation** to instrument your unmanaged code.

◇ Select **DevPartner > Native C/C++ Instrumentation Manager** to open the Instrumentation Manager dialog box to make additional adjustments to instrumentation of your unmanaged code.

5   Choose **DevPartner > Start Without Debugging with Performance Analysis**, or click **Start Without Debugging with Performance Analysis** on the DevPartner toolbar to begin your analysis session.

◇   DevPartner cleans and rebuilds your unmanaged code with instrumentation, then starts your application.

◇   DevPartner instruments managed code applications as they execute.

6   When you are finished collecting data, quit your application.

Controlling Data Collection

DevPartner gives you three ways to control when performance data is collected during the use of your application:

◆   You can use Session Control icons on the DevPartner Session Controls toolbar to interactively control data collection as your program runs.

◆   You can use the Session Control API to control data collection in your program.

◆   You can use the Session Control file to assign Session Control actions to specific methods in your application modules.

For information on using Session Controls, see the Performance Analysis online help in the Visual Studio Combined Collection.

Viewing Your Results

This section provides a brief description of the DevPartner windows and dialogs that you can use to work with performance analysis session data. See the DevPartner Performance Analysis online help in the Visual Studio. NET Combined Collection for more information.

Session Window

When you quit your application, DevPartner displays the performance data in a Session window in Visual Studio. The Session window contains the Filter pane and the Session Data pane.
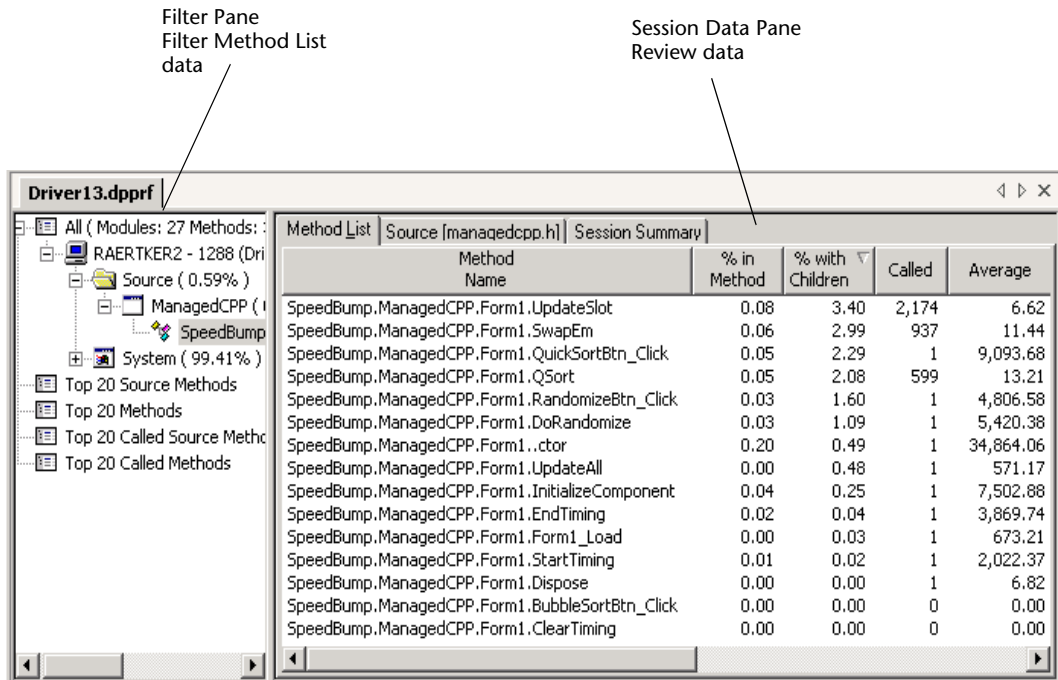
Filter Pane
Filter Method List
data

Session Data Pane
Review data

Method Name	% in Method	% with Children	Called	Average
SpeedBump.ManagedCPP.Form1.UpdateSlot	0.08	3.40	2,174	6.62
SpeedBump.ManagedCPP.Form1.SwapEm	0.06	2.99	937	11.44
SpeedBump.ManagedCPP.Form1.QuickSortBtn_Click	0.05	2.29	1	9,093.68
SpeedBump.ManagedCPP.Form1.QSort	0.05	2.08	599	13.21
SpeedBump.ManagedCPP.Form1.RandomizeBtn_Click	0.03	1.60	1	4,806.58
SpeedBump.ManagedCPP.Form1.DoRandomize	0.03	1.09	1	5,420.38
SpeedBump.ManagedCPP.Form1..ctor	0.20	0.49	1	34,864.06
SpeedBump.ManagedCPP.Form1.UpdateAll	0.00	0.48	1	571.17
SpeedBump.ManagedCPP.Form1.InitializeComponent	0.04	0.25	1	7,502.88
SpeedBump.ManagedCPP.Form1.EndTiming	0.02	0.04	1	3,869.74
SpeedBump.ManagedCPP.Form1.Form1_Load	0.00	0.03	1	673.21
SpeedBump.ManagedCPP.Form1.StartTiming	0.01	0.02	1	2,022.37
SpeedBump.ManagedCPP.Form1.Dispose	0.00	0.00	1	6.82
SpeedBump.ManagedCPP.Form1.BubbleSortBtn_Click	0.00	0.00	0	0.00
SpeedBump.ManagedCPP.Form1.ClearTiming	0.00	0.00	0	0.00

Figure 6-1. The DevPartner Session window displays performance analysis data

The Filter Pane

The **Filter** pane lists the source files and system images used during the session. The percentage that appears to the right of each name identifies the time spent in that file as a percentage of the total session time. Click items in the **Filter** pane to select the subset of session data you want to view in the **Method List**. For example, selecting a source file in the **Filter** pane displays the methods in that file on the **Method List** tab.

Note: Method refers to the methods, functions, and procedures used by your application. System refers to any non-instrumented module used by the application.

The Session Data Pane

The **Session Data** pane contains the **Method List**, **Source**, and **Session Summary** tabs.

◆ The **Method List** tab shows performance data for the file, or group of files, that you select in the Filter pane. By default, the application executable is selected, and the Method List tab displays all application methods that were used during the session.

You can click a column header to sort the data by that column. Click the column header again to reverse the sort order.

◆ The **Source** tab displays the contents of the currently selected source file. Data columns to the left of the code provide line-by-line performance data. For example, the Count column indicates how many times each line was executed during the session. DevPartner also highlights the slowest line in each method with the system-highlight color.

Double-click any source file in the Filter pane to switch to the Source tab and display the contents of the file.

◆ The **Session Summary** tab displays summary information about the session, as well as a description of the system and operating environment in which the session occurred.

Comparing Sessions

DevPartner gives you the ability to compare performance sessions. When you compare sessions you can see at a glance the impact of optimizations you make on individual methods and on application performance as a whole.

Naturally, comparisons are most useful when you compare session files created by exercising the application identically. For example, do not compare sessions run without debugging to sessions run in the debugger. You can compare sessions if you have captured two or more session files.

Run your application under performance analysis to create a session file.

Note the most expensive methods and optimize one of the expensive methods.

Run a second, identical, test of your application under performance analysis.

In the second or current session window, right-click and select **Compare** from the context menu. Choose the first session you ran as the basis session.

Compare the session data for the methods you optimized and other methods whose performance values changed between sessions.

Before and after graphs for key Method List columns

Total clock time difference between sessions

Browse to select a new basis session

Place cursor over column in the "before and after" graphs to view details for methods in a ToolTip

Place cursor over a method in the Method List to view performance comparison details
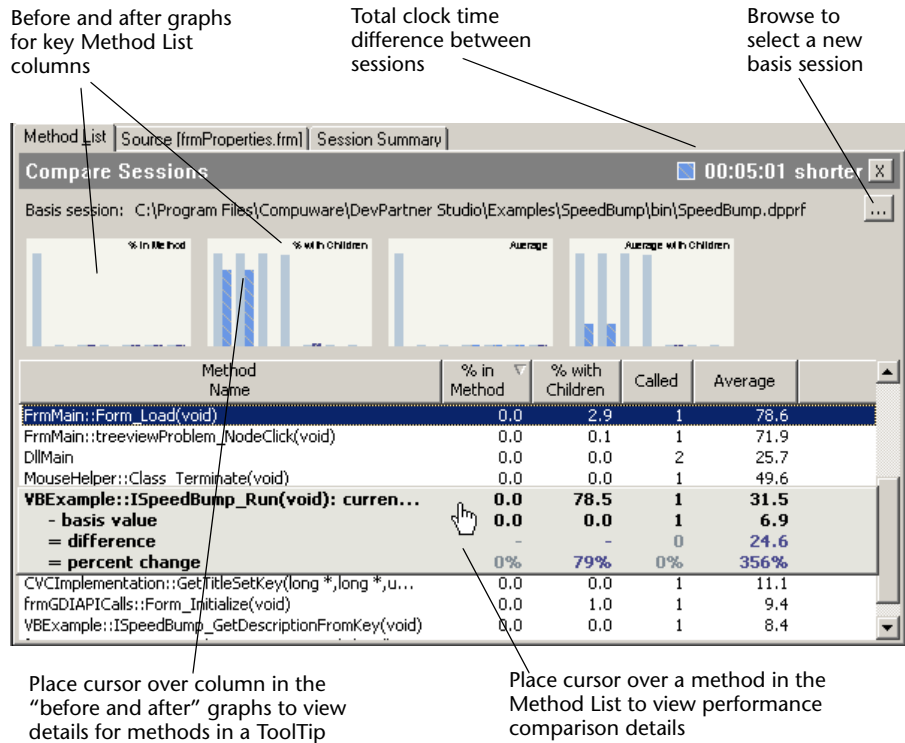
Figure 6-2. Comparing Performance Sessions

When you finish comparing, press **Esc**, or click the Compare icon on the DevPartner Performance Session toolbar.

Interpreting Session Comparison Results

A session comparison shows the **current value**, **basis value**, **difference**, and **% difference** between a method in the current session and the same method in the basis session. DevPartner uses color to help you see at a glance whether the value in the current session is larger or smaller than that in the basis session. When the values for difference and % difference are dark blue, the values for the current session were better (faster) than those of the basis session. Light blue means that the performance values were slower in the current session.

Once you have determined what results your code changes accomplished between sessions for any given method, search other methods in the session to uncover any side effects of your initial code changes. Even though an individual method's performance improved, the larger

program's performance may have degraded. In performance tuning, no tool can substitute for thorough knowledge of the structure of your code.

When examining session comparison results, be aware of the following:

◆ A percentage is a ratio of two numbers. Percentages are additive only when computed relative to the same total value.

◆ If one percentage value decreases, all other percentage values must increase. In a complex program this may be difficult to notice, since the percentage increase must be averaged across all the other methods in the program.

◆ To interpret a subprogram's timing, you must understand that subprogram's role in the enclosing program.

◆ Performance measurements have no meaning outside the context of the program that produced them. It is not possible to generalize about the effects of program changes without understanding the program's operation.

Once you are satisfied with the changes to the costliest method in your program, you can turn your attention to other expensive methods.

The Call Graph

Use the **Call Graph** to analyze method performance in depth. You can access the Call Graph from the Method List tab or the Source tab.

◆ On the **Method List** tab, right-click a method name and select **Go to Call Graph** on the context menu to display the Call Graph for the selected method.

◆ On the **Source** tab, right-click any line of code that executed during the current session. On the context menu, select **Go to Call Graph** to display the Call Graph for the method that contains the selected line.

The Call Graph shows the selected method and the methods with which it has a call relationship. The selected method appears as the base node in

the graph. Parent methods (nodes) appear to the left of the base node, and child methods appear to the right.
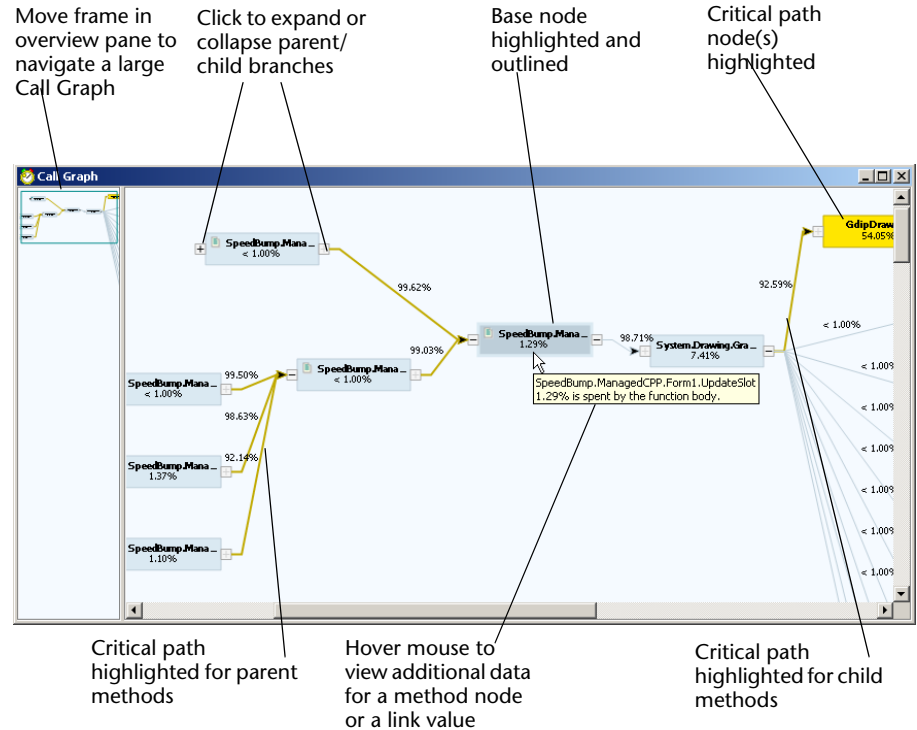
Figure 6-3. Using the Call Graph

In the Call Graph:

- Click the plus/minus icons at the edges of any node to expand or collapse the view of parent (left) or child (right) nodes.

- Right-click any node and choose **Go to Method Source** to view source code for the method.

- Hover the mouse-pointer over a node or over the percentage value on a link between method nodes to view a more detailed description.

- To view a Call Graph with a different base node, close the Call Graph window and right-click a different method in the Method List.

Using the Call Graph

When you run a performance analysis session, scan the Method List for methods with high values in the % in Method and % with Children columns. Display the Call Graph using that method as the base node.

Child-side Analysis	Analyze the child (right) side of the Call Graph to understand what to optimize.

Expand the child nodes to analyze whether the base method or a child method is responsible for the most time.

◆ If the base node has several parallel branches, look for branches that have the largest values on the link to the first child method. Optimizing branches with higher values is likely to provide more benefit in terms of performance.

◆ If the base method itself shows a high value, consider optimizing the base method.

◆ If a child branch is a large contributor to the time spent by the base method, look for child nodes with high percentage values.

Parent-side Analysis	Analyze the parent (left) side of the Call Graph to determine if the base node branch is worth optimizing, or if it is feasible to eliminate or reduce the number of times the base node is called.

Expand the parent nodes to the left of the base node. In particular, examine the base node's contribution to the time spent in its parent branches. This will help you determine if optimizing the base node or its child methods is worthwhile. If the base method is a large contributor to several parents, or to an important parent in terms of overall program execution, it is probably worth considering as a target for optimization.

Note: Values on the links between the base node and its parents are independent, not additive. Each percentage value represents the base node's contribution to the time spent by that parent.

◆ If the base node has several parents, and one or more values on the links to the base node is high, the base node may be a candidate for optimization.

◆ If the values on the links to the parents are very small, optimizing the base node branch will probably have little impact on parent method performance.

◆ To determine if the base node is the best choice to optimize, view a new call graph with the parent selected as the base node. This will show the importance of the original base node to the parent node's performance, relative to other children of that parent method.

When analyzing either the parent or child side of the Call Graph, you can right-click a node at any time and use the context menu to view the source code for the method to see if you can determine why it is using so much time.

To Display a Selected Source File

In the **Filter** pane, double-click the source file.

To Display Source Code for a Selected Method

1   Select a method name.

2   Right-click, and on the context menu, select **Go to Method Source** or **Go to Child Source**.

Complete the following steps to display a specific method on the Source tab.

1   Click the **Source** tab.

2   Click the **Select Source Method** list on the Performance Session toolbar.

3   Select the name of the method you want to view.
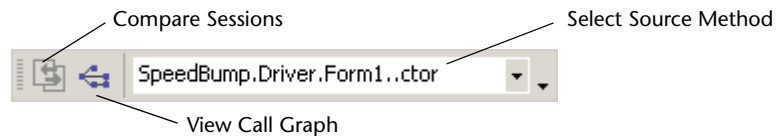
Figure 6-4. Performance Session toolbar

Performance Analysis for Real World Application Development

Application performance depends on several factors, some external, such as CPU speed, others internal, such as the algorithms and logic used in coding the application. Despite constantly escalating hardware capabilities, developers cannot escape the need to improve the performance of their code if they want to produce competitive applications.

Finding Bottlenecks

The key concepts for effective performance analysis can be summed up simply:

◇   Measure performance to identify bottlenecks
◇   Modify code
◇   Measure again to verify impact

In many applications, a relatively small portion of the code is responsible for much of the application's performance. In order to improve performance of critical parts of the code, use DevPartner performance analysis to locate performance bottlenecks, and subsequently, to verify that the improvements you make really do impact performance.

If you are dealing only with code you wrote, locating and fixing a performance problem is relatively straightforward: Change the application logic or use a different algorithm. However, in modern software development, only a fraction of the code that runs when your application executes is likely to be your own. This is especially critical to remember when analyzing .NET applications.

Effective Performance Analysis for .NET Applications

The .NET Framework is particularly rich and complex. You can accomplish a lot with a few lines of code. This offers great opportunities to developers, but can make it difficult to tune application performance. For example, you may discover that 95% of your application's execution time is spent in the Framework. How do you improve performance in that case?

Here are some basics to make the performance analysis process more productive.

◆ Analyze source code

Use the **Top 20 Source Methods** filter to isolate application hotspots.

Use the Call Graph to examine the most expensive methods to understand the costs associated with child methods called.

Compare the effect of different algorithms or logic changes by running multiple performance sessions.

◆ Understand Framework costs

Use **% with Children** on the Method List or Source tab to see how much time you are spending in the Framework.

Drill into the Framework by examining child methods in the Call Graph to understand which calls are expensive and why.

Rework the application to do less work or to call the Framework less often.

◆ Understand start-up costs

Use the **Clear** Session Control before collecting performance data. The .NET Framework performs many one-time initializations. To prevent these from skewing performance results, warm up the application by exercising all the features you want to profile, then

*Tip: To avoid collecting data for all system (non-source) files, check **Exclude system images** on the DevPartner **Exclusions - Performance** options page. Once you optimize your source code, turn off this option so you can examine how your application uses system code, especially the .NET Framework.*

Clear the data. Next, run a test that exercises the same features to get a more accurate performance picture.

◆ Understand what you want to measure

Consider how your application behaves before you begin collecting performance data. For example, if you are profiling a Web services or ASP.NET application, think about how Web caching will affect your results. If your test run inputs the same data repeatedly, your application will fetch pages from the cache, skewing the performance data. In such a case, you could take pains to insure variable input data, or simpler, edit the `machine.config` file to turn off caching while you test. Comment out the line that reads:

```
<add name="OutputCache"
type="System.Web.Caching.OutputCacheModule"/>
```

◆ Measure performance of mixed-mode applications

You may choose to write parts of a .NET application in native (unmanaged) C/C++. DevPartner allows you to collect performance data for both managed and native portions of an application in a single run, provided the native code is in a separate file and you instrument the code before collecting data. Thus, you can compare the effectiveness of native and managed code in the context of the total application by comparing performance sessions.

◆ Collect complete data for distributed applications

When you analyze performance for a Web application, a multi-tier client/server application, or an application that uses Web services, include all remote application components in the analysis. Use a DevPartner installation to configure performance data collection on remote systems. If your application uses native C/C++ components, instrument the components for performance analysis before collecting data. Of course, the recommendations regarding start-up costs, Framework costs, and awareness of application behavior apply equally to collecting data for server-side components.

◆ Understand the limitations of micro-profiling

Once you identify a bottleneck in your application, you may find it convenient to create a smaller sample of code that duplicates the problem area in the main application. You improve performance in that sample by iterative performance comparisons and then move the code back in to the main application. Is your application going to be faster? Maybe. But you cannot know until you rerun your original performance tests.

Extend this idea. Modern software development is a team enterprise. You regularly analyze the performance of your part of the application

and make performance improvements. Your colleagues do the same for the components on which they work. When you put the pieces together, you are going to have the fastest application on the planet. Application memory footprint, multi-threading, thread priorities, process security, network latency, server load, and other contingencies can affect the way your code runs in ways you may not foresee, and more important, in ways that performance testing of a single component may not reveal. You have not measured application performance until you have simulated as closely as possible the conditions under which your application is going to be used.

Running a Program from the Command Line

In addition to running your program within Visual Studio, you can use command line executables to collect performance profiling information without launching Visual Studio.

For Visual C++ and Visual Basic applications, running your program from the command line is the only way to collect performance data.

The following resources provide details about running your program from the command line:

◆ Refer to the DevPartner Performance Analysis online help in Visual C++ 6.0 and Visual Basic 6.0.

◆ Refer to the section *Command Line and Configuration File Usage* in the DevPartner Studio online help within the Visual Studio combined help collection in Visual Studio 2003 or Visual Studio 2005.

Analyzing Performance in Visual C++

You can use DevPartner to instrument your Visual C++ applications. Follow this workflow to analyze them:

◆ From the **DevPartner** menu in Visual C++, select one of the following commands to instrument your application:

  ◇ **Build > Performance** to instrument only the new or changed parts of the active project.

  ◇ **Rebuild All > Performance** to re-instrument the entire project.

  ◇ **Batch Build > Performance** to instrument or re-instrument selected files and their dependencies.

- Use `DPAnalysis.exe` to run your application and generate a session file. You can either:
  - ◇ Use `DPAnalysis.exe` to run your application directly from the command line.
  - ◇ Use `DPAnalysis.exe` to execute an XML configuration file to run your application.

- To analyze your results, use Visual Studio 2003 or Visual Studio 2005 to open the session file.

For more information, refer to the DevPartner performance analysis online help in Visual C++ 6.0 and the section *Command Line and Configuration File Usage* in the DevPartner Studio online help within the Visual Studio combined help collection in Visual Studio 2003 or Visual Studio 2005.

Analyzing Performance in Visual Basic

You can use DevPartner to instrument your Visual Basic applications. Follow this workflow to analyze them:

- On the Visual Basic **DevPartner** menu, select **Make with Performance Analysis** to instrument the active project without running it

- Use `DPAnalysis.exe` to run your application and generate a session file. You can either:
  - ◇ Use `DPAnalysis.exe` to run your application directly from the command line.
  - ◇ Use `DPAnalysis.exe` to execute an XML configuration file to run your application.

- To analyze your results, use Visual Studio 2003 or Visual Studio 2005 to open the session file.

Chapter 7
In-Depth Performance Analysis
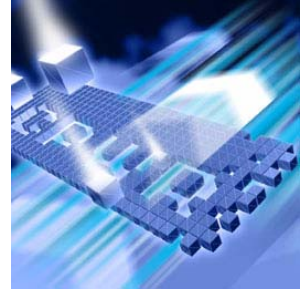
- ◆ What is Performance Expert?
- ◆ What Can I Do with Performance Expert?
- ◆ Who Should Use Performance Expert?
- ◆ Finding Application Problems with Performance Expert
- ◆ Usage Scenarios
- ◆ Automating Data Collection
- ◆ Collecting Data from Distributed Applications
- ◆ Performance Expert in the Development Cycle

What is Performance Expert?

DevPartner Studio contains many features designed to assist application development, including a performance analyzer that helps you locate bottlenecks in your code. Performance Expert takes performance profiling a step further for managed code Visual Studio applications by providing deeper analysis of the following hard-to-solve problems:

- ◆ CPU/thread usage
- ◆ File/disk I/O
- ◆ Network I/O
- ◆ Synchronization wait time

Note: Performance Expert analyzes managed code only, and is therefore not supported in the DevPartner for Visual C++ BoundsChecker Suite.

Performance Expert analyzes your application at run-time and locates the problem methods in your code. It then allows you to view details about individual lines in the method, or to examine parent-child calling

relationships to help you determine the best way to fix the problem. When you have decided on an approach, Performance Expert enables you to jump directly to the problem lines in your source code, so you can quickly fix problems.

DevPartner Performance Expert is designed for use by software designers, software developers, and quality assurance (QA) engineers. It can also be used by development management staff to identify problems in an ongoing project.

Performance Expert and Performance Analysis

Think of Performance Expert as a complement to traditional performance profiling. Use it in conjunction with DevPartner's performance analysis feature. Run performance analysis to get a baseline view of application performance. Use Performance Expert to better understand the nature of difficult problems, especially problems that involve disk or network I/O, or synchronization issues. When you have fixed the problem, run the application in a performance analysis session and use the performance analysis Session Comparison feature to verify the improvement.

What Can I Do with Performance Expert?

You can use Performance Expert to improve performance of any managed code Visual Studio application, including:

◆ ASP.NET Web applications

◆ ASP.NET Web services applications

◆ .NET Remoting server applications

◆ Windows Forms client applications

◆ Serviced components, e.g. COM+

Typical target applications include WinForms client applications and ASP.NET or other Web applications, but you can analyze any managed code application, as the following examples illustrate.

WinForms Client Application

Presentation layer, business logic, and analysis functions execute in a single process on a WinForms client. The client accesses data via a traditional database call using OLEDB or ADO.NET, or via a SOAP call to a

Web service that executes the database call and possibly performs other actions.

Distributed Web Application

In a distributed Web application that uses ASP.NET technology, the presentation layer is dynamically generated on the Internet Information Services (IIS) server for rendering in the browser (Internet Explorer) client. The business logic uses SOAP or .NET remoting to access a Web service and various managed components. The Web service or a managed component handles calls to the database.

Who Should Use Performance Expert?

DevPartner Performance Expert will be most useful to software developers and designers, but many members of the engineering team can use the feature at several points in the software project life cycle.

Software Designer

Software designers must often develop prototypes that meet specific requirements, for example, in response time or scalability. Before producing the final design, the designer must identify the operations and, if possible, the methods, that are preventing the prototype from meeting the performance requirements. Ideally, the designer would like to be able to identify a few methods that, if fixed, would give a dramatic performance boost.

Software Developer

The software development team builds the application based on the designer's prototype and specification. As soon as the application (or application components) can be tested and run, developers can integrate Performance Expert into their automated testing routines in order to identify potential CPU usage, file I/O, or network I/O issues as they are coding and debugging. Developers can review the Performance Expert session log each morning to see if the previous day's coding has introduced any new performance issues and address issues immediately. When coding is complete, the development team submits the final Performance Expert session log to document that performance goals have been met.

Quality Assurance

Quality Assurance teams can use Performance Expert to continuously monitor application performance. QA can easily integrate Performance Expert into automated test suites to obtain a daily reading of application performance in critical areas. When problems appear, QA teams can send the session log to the development team or attach the log to a bug report in a defect tracking system such as Compuware TrackRecord.

Finding Application Problems with Performance Expert

DevPartner Performance Expert helps you identify problems in the following critical areas:

◆ CPU/thread use (including wait and synchronization issues)

◆ File and disk I/O

◆ Network I/O

◆ Synchronization wait time

When run from Visual Studio, Performance Expert analyzes a single process at a time. It reports data for any threads executing in the selected process. To analyze an additional process, select the second process and rerun Performance Expert. Performance Expert can also analyze a distributed application that spans multiple machines. For information about remote data collection, see "Collecting Data from Distributed Applications" on page 167.

Basics: Running a Performance Expert Session

To locate problems in your application:

1   Open the solution that contains your application in Visual Studio.

2   Turn on Performance Expert data collection by choosing **Start Without Debugging with Performance Expert** on the DevPartner menu, or by clicking the Performance Expert icon on the DevPartner tool bar.

3   Start your application and exercise the slow portion.

4   Watch the **Performance Expert** window to monitor your application. A spike in the CPU, disk, or network activity indicates a potential trouble spot.
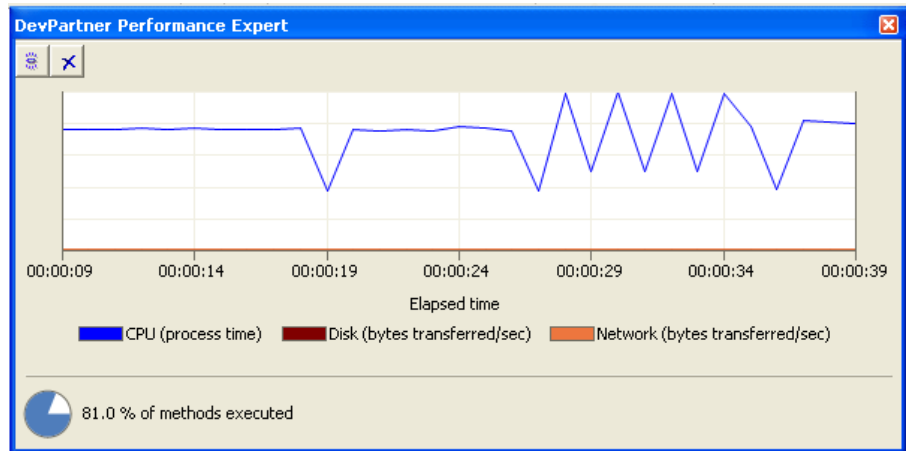
Figure 7-1. Control Data Collection with the Performance Expert Window

5   When you have finished collecting data, stop your application. Dev-Partner generates a Performance Expert session file.

6   As an alternative, you can use the **Snapshot** and **Clear** session controls at the upper left of the **Performance Expert** window to manage data collection as your application runs.

Note: Generally, run Performance Expert sessions without debugging. Although DevPartner collects accurate performance data with or without debugging, results from non-debug sessions are easier to interpret and do not include the processing overhead caused by the debugger. In non-debug sessions, you get the results faster and they are not skewed by differences in debugger overhead for different types of code. For example, exception handling code takes longer relative to other parts of the application in the debugger than it does without debugging. If you run your application in the debugger, some timing values might be larger than expected, especially if breakpoints were hit during the session. Expect tracing and other debug-only functionality to figure highly in such session files.

The Session File

Tip: An entry point method is a source code method that was not called by another source code method, i.e., an entry point into source code execution.

In the session file results summary, examine the two bar graphs. The **Paths that used the most CPU** graph shows the entry point methods for the most expensive calling paths that Performance Expert measured as you ran your application. The **Individual methods that use the most CPU** graph focuses attention on methods that are expensive in themselves, exclusive of source code child methods they call.

Paths that use the most CPU

Method	CPU time
Form.Main	10,418,100.0
Form.CtoF	9,815,903.0
Service.CtoF	7,426,693.0
Service..ctor	2,386,896.0
Form.ParseOption	163,456.4
Form.FtoC	26,761.4

0.0   CPU time in method (microseconds)   10,418,110.0

Individual methods that use the most CPU

Method	CPU time
Service.CtoF	7,426,693.0
Service..ctor	2,387,093.0
Form.Main	411,780.0
Form.ParseOption	163,658.9
Service.FtoC	25,747.0
Form.CtoF	2,314.6

0.0   CPU time in method (microseconds)   7,426,698.0

43.8 % of methods executed   Total elapsed time: 10,418,100.0 µs   Total execution time: 10,418,100.0 µs

Figure 7-2. The Performance Expert Session File Shows the Paths and Methods that Consumed the Most CPU

Click a method in either graph to view details for the method. If you drill down from the **Paths that used the most** CPU graph, you can view **Call Graph** and **Call Tree** presentations of the session data. The **Call Graph** shows the child methods called by the entry point method, with the relative contributions to the time spent in the path. The **Call Tree** presents a tree view of the same data and enables you to view additional data about each method. If you choose to examine a method in the **Individual methods that use the most CPU** graph, DevPartner presents a **Methods** table of your application's methods with additional data to assist your troubleshooting. To switch between the **Paths** and **Individual methods** views, click **Back to Summary** in any details view.

Accounting for Child Methods

The calculation of the Performance Expert session data differs between the Paths and the Individual methods views. DevPartner excludes measurements for source code child methods in computing data for CPU time, disk or network I/O, and synchronization lock wait time in the Individual method analysis views. In contrast, DevPartner includes the impact of source code child methods to their parent methods in the Path analysis views.

All computations in both views include time or throughput attributable to system or .NET Framework methods called by your source code

methods. Managed applications typically spend a lot of time executing Framework code. Performance Expert charges the system data to the lines in your source code that made the calls in order to focus attention on how your code interacts with the Framework, that is, on the parts of the application that you can modify.

For detailed information on additional features you can use to control data collection, and tips on analyzing the session data, see the Performance Expert online help.

Usage Scenarios

The typical methodology for resolving performance issues consists of the following steps.

1   Locate the slowest line in a problem method and optimize it.

2   If you cannot optimize the line, remove it or execute it less often.

In the simplest cases, you may be able to locate the slowest line in a method (e.g., by using the DevPartner Performance Analysis feature) and either optimize it or call it less often. However, in real world application development, many problems have more complex causes. You may be able to identify the slowest method, only to find that a combination of lines within the method is slowing execution. In such a case, additional targeted data can help you analyze the problem quickly.

For example, if the slowest part of your application does a lot of network I/O, the following metrics would likely help you understand the nature of the problem:

◆   Total number of network reads and writes

◆   Number of bytes read or written

◆   Number of read or write errors

◆   Elapsed time for read or write operations

If your application did a lot of disk I/O, you would want to see metrics that reflected read/write volume and the efficiency of those operations. DevPartner Performance Expert reports exactly this kind of data.

You can use DevPartner Performance Expert to analyze application performance in the areas of CPU and thread performance, disk I/O, network I/O, and synchronization wait time. The following examples will illustrate ways in which you can use Performance Expert to improve application performance.

Identifiable Performance Problem

Usability testers have reported that specific operations in your application are too slow. As a developer, you want to locate the parts of your source code that are responsible for the slow operations taking so long to complete and fix them.

Let's assume that you have run the slowest part of your application under Performance Expert as described in "Basics: Running a Performance Expert Session" on page 156. When you examine the session file, you immediately see the method that took the longest time to execute at the top of the **Individual methods that use the most CPU** graph. However, in a complex application, a single slow method may affect performance less than a sequence of moderately slow methods. The slowest calling sequences appear in the **Paths that use the most CPU** graph. Do some methods appear in both graphs? If so, these methods definitely deserve scrutiny.

You also notice that some of the methods in the graphs are marked with icons that indicate disk I/O or network I/O activity in the method. These indicators tell you something about the kind of processing done by these methods.

- Disk activity
- Network activity

At the bottom of the summary view, Performance Expert displays the **Total elapsed time** and **Total execution time.** If the execution time is very small relative to elapsed time, and you have exercised the application in such a way that you are reasonably sure the difference is not simply due to waiting on user input, check to see if some methods in your application are spending more time waiting for locks than they should.

Let's say you first decide to examine the top method in the **Individual methods that use the most CPU** graph. You understand that many factors can affect CPU utilization: processor-intensive computations, disk I/O, network I/O, or inefficiently used synchronization objects. Similarly, you know that wait time can have multiple causes: the resource your method is waiting for could be shared within the same process, or with an external process. But how do you quickly determine what is going on in your application?

Click the top method in the **Individual methods that use the most CPU** graph to open the **Methods** detail view for the method. Notice the data in the columns in the **Methods** table. This information should help you determine what the method is doing. If the method was marked with the disk activity icon in the graph, right-click in the table and use

the **Choose Columns...** dialog to add all of the disk-related columns to the table. You might find that the method is producing read or write errors, or is using a large amount of time to write small amounts of data, and is being executed many times.

The **Source** tab in the lower half of the **Methods** window shows you the source code for any method you select in the table. When you click on a method in the table, the source automatically scrolls to the line that consumed the most CPU time and indicates the time attributable to that line. The view also indicates graphically other lines that used CPU time.

If the method performed disk or network I/O, or had wait time, expanding the list at the upper left shows those selections, so you can immediately locate the most significant line in the method for that metric. For example, Choose **Disk activity** from the drop-down list to immediately go to the line that transferred the most bytes, and to see relative disk activity for other lines in the method. If the method involves **Wait time**, check that view too. Notice which lines are associated with long wait times. In each view, DevPartner selects the most expensive line by default. Comparing these views of the lines in the method shows you where to focus your efforts much more quickly than traditional debugging techniques.
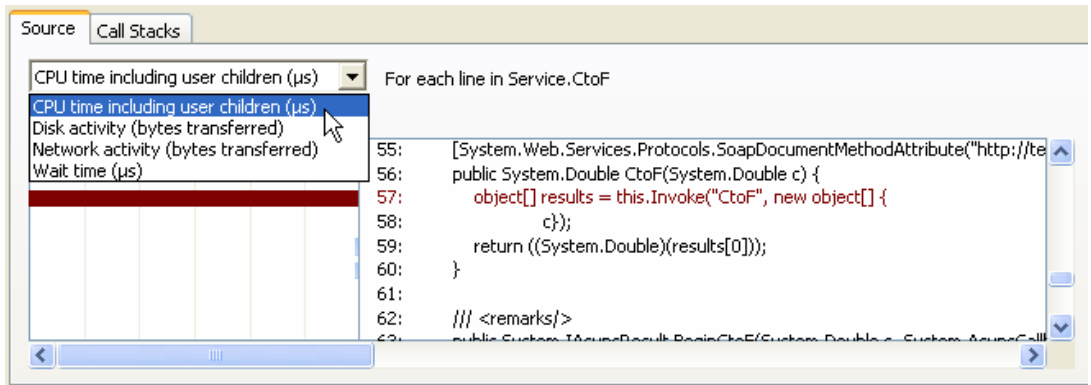
Figure 7-3. The Source Tab Helps Locate Problem Lines in a Method

When you have located an appropriate line to fix, double-click on it to jump to that line in your source code in Visual Studio.

If a way to fix the problem is not obvious, click the **Call Stacks** tab to see all the ways the method was used as your application executed. Is the problem method called by more than one path? If so, examine the call stacks that are responsible for the most time in the method.

Obviously, you want to look first at the parent path responsible for the highest percentage of calls. Try to modify your code to eliminate the calls, or call less frequently. The **Call Stacks** tab includes a view of your source code. When you select a method in the stack, the source automatically scrolls to the line where the call to the next method in the stack was made. A double-click opens the line in Visual Studio, so you can quickly modify the calling sequence if necessary. Once you have made the changes to your code, run the application again with Performance Expert to verify the improvement.

Scaling Problem in an Application

Your new Web application runs fine when you test it on your machine. But when you allow additional users to access the application, it is too slow. You have a looming deadline. How do you quickly determine what is wrong?

You can collect Performance Expert data while stressing your application with a load-testing tool. To do so, you will probably want to start and stop your application with a command line tool or script. DevPartner provides a command line utility called `DPAnalysis.exe` for this purpose. For information on running a Performance Expert session from a command line, see . For example, you could do something like the following:

1 Start the application under Performance Expert with `DPAnalysis.exe`.

2 Run the load-testing application.

3 Stop the application.

4 Examine the Performance Expert session data.

Let's assume that when you look at the session file, no single method in the **Individual methods that use the most CPU** graph stands out as the likely culprit. It is a complex application, and it is probable that several methods contribute to the sluggish performance. Start your analysis with the **Paths that use the most CPU** graph in the summary view. This graph shows a list of methods, but in this case each method represents an **entry point**. An entry point method is a method that was not called by another source code method, that is, an entry into code that you wrote, and more importantly, that you can change. The entry point method that corresponds to the most expensive path of execution in your application appears at the top of the graph. Click on the bar to open the **Path analysis** view.

The Call Graph

The **Path analysis** view consists of a **Call Graph** that shows the child methods called from the selected entry point method, and a **Call Tree** view that presents the data in tree form in a table with columns of data for each method in the application. When you open the **Call Graph** from the results summary, notice that DevPartner places the most expensive paths at the top of the **Call Graph**, and highlights the most expensive child path whenever a path branches. As you examine the data, investigate the most expensive child paths first. To investigate a path, expand the nodes to the right.

Tip: The percentages on lines connecting a method to the child methods it called are additive; those on lines connecting the chain of methods in a single path are not.

To determine the relative contributions of different paths spawned by the same method, compare the percentage values on the lines that connect the selected method to each of its child paths. The value on each link represents the percent of time in the parent method attributable to child methods called in that path. Thus, in Figure 7-4, the method Form.Main called Form.CtoF, Form.ParseOption, and Form.FtoC. The value on the line that links Form.Main to Form.CtoF is 98.1%, while the remaining 1.9% is spread among the other called paths. This means that the path Form.Main calls Form.CtoF accounted for 98.1% of the CPU time spent in Form.Main that was attributable to the execution of child methods. Start your troubleshooting with this path.
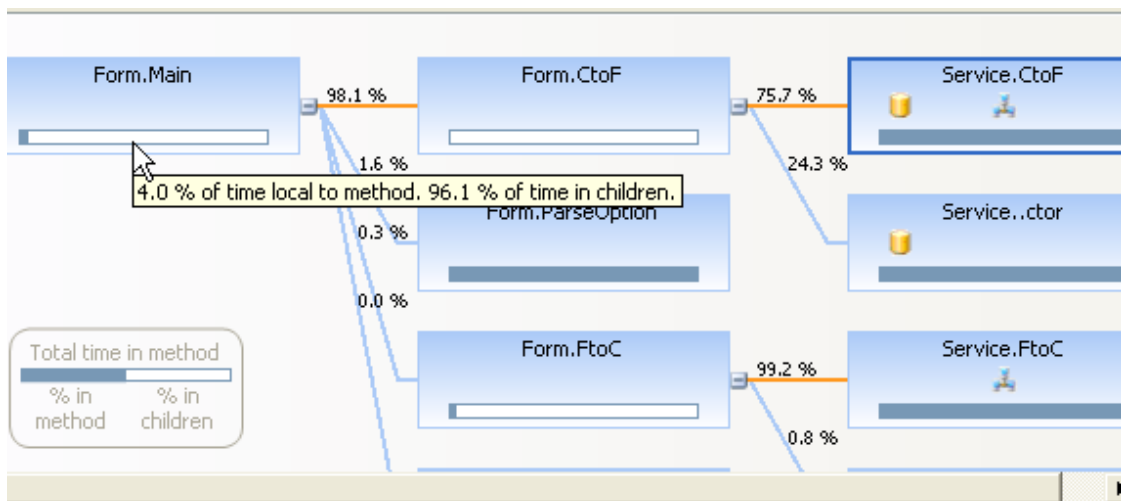
Figure 7-4. The Call Graph Shows the Impact of Child Methods

As you investigate the called path, notice the horizontal bar at the bottom of each node. The bar shows the relative percentages of time in the method due to the method body compared to the child methods it

called. Hover over the bar with the mouse to see the actual percentages. Use this bar to guide your tuning efforts. For example, if 4% of time is spent in the method body, and 96% of time is attributable to child methods, continue to investigate the most expensive called paths to locate the child methods that are affecting performance. Fix those methods or change your code so they can be called less often. If, on the other hand, 96% of the time was spent in the method body, focus your efforts there.

Also notice whether an expensive node contains the disk activity, network activity, or wait time icons. Hover over the icon with the mouse to view the magnitude of the activity. If a node contains one or more of these icons, consider switching to the **Call Tree** view and adding the appropriate data columns for more help in diagnosing the problem.

The Call Tree

The default sort of the **Call Tree** table is by **CPU time including user children**. To gain an idea of where the bulk of the time is being spent, scan the values in the other columns. Doing so will tell you whether wait time, disk or network I/O, or CPU-intensive processing is the major factor. If you need more detail, you can add additional columns, such as disk or network reads, writes, and errors, to the display.

Figure 7-5. The Call Tree Displays Additional Data for the Selected Method

For example, if an expensive method in the **Call Graph** showed network I/0, select it, switch to the **Call Tree**, and add all of the network-related

data columns to the table. To add columns, right-click in the **Call Tree** table and select **Choose columns...** from the context menu. For a full explanation of the data reported in each column, see the Performance Expert online help.

Whether you are using the **Call Graph** or **Call Tree**, the session file window includes the **Source** and **Call Stacks** tabs. These tabs function as they do in the Methods view, except that the data is calculated to include data attributable to user, or source code, child methods. Use the **Source** tab to immediately locate the most expensive line in any method you select in the **Call Graph** or **Call Tree**. Use the **Call Stacks** tab to see the relative impact of other paths that called the method and to locate the line that called the selected method in the stack. Double-click a line of code in either the **Source** or **Call Stacks** tab to jump to that line in Visual Studio for editing.

Performance Slow but No Specific Issue

Suppose your application is generally sluggish, but you cannot identify a specific issue. Performance tuning is an iterative process. You can still use the techniques described above to try to improve performance.

◆ Run the application under Performance Expert.

◆ Go through the **Paths that use the most CPU** and try to optimize the most expensive branches for each critical path.

◆ Go down the list of **Individual methods that use the most CPU** in the same way and try to optimize the top methods in the list.

◆ Retest to verify improvement.

Automating Data Collection

DevPartner Performance Expert supports command line execution through an executable called `DPAnalysis.exe`. This file is located in your `\Program Files\Compuware\DevPartner Studio\Analysis\` directory. You can run an application under Performance Expert from a command prompt, or create batch files to automate data collection. You can launch the Performance Expert session in two ways:

◆ Specify the target and arguments in standard MS-DOS command line syntax

◆ Specify an XML configuration file that contains the targets and arguments for the session

Consider the example we discussed in the section "Scaling Problem in an Application" on page 162. Quality Assurance engineers can monitor scalability (or any other aspects of the application) on a daily basis by setting up an automated test (or suite of tests) to be run on the application every night. To automate the tests, set up a batch file to

1 Start the application under Performance Expert

2 Start the load-testing application and any other tests you want to run

3 Stop the application when the tests are complete

DevPartner automatically generates the session log file when the application exits.

The command line syntax to launch the session is:

```
dpanalysis /Exp /E /O /W /H [/P or /S] target {target
arguments}
```

/Exp Sets analysis type to DevPartner Performance Expert

/E Enables data collection for the specified process/service

/O Specifies the session file output directory and/or name

/W Specifies the working directory for the process

/H Specifies the host machine on which the target runs

/P or /S specifies that the target is a process or a service; use only one

There is one restriction on the order in which the switches must appear: The /P or /S switch must occur last. Everything that follows either switch is interpreted as an argument to the process or service.

To use an XML configuration file, the command line is even simpler: `dpanalyis /C [path]configuration_file.xml`.

The configuration file contains the necessary parameters for any type of DevPartner analysis, including some options that are not available using command line switches. For example, if you want to exclude application components from a Performance Expert session, you must use the `ExcludeImages` element in the configuration file.

To collect data for a process that runs on a remote machine, you must specify a directory and file name. Use the `SESSION_FILENAME` and `SESSION_DIR` elements in the Analysis options in the configuration file.

```xml
<?xml version="1.0" ?>
<ProductConfiguration xmlns="http://www.compuware.com/products">
        <RuntimeAnalysis Type="Expert" MaximumSessionDuration="1000"/>
        <Targets RunInParallel="true">
                <Process CollectData="true" Spawn="true" NoWaitForCompletion="false">
                        <AnalysisOptions NO_MACH5="1" NM_METHOD_GRANULARITY=""
                        SESSION_DIR="c:\Sessions" SESSION_FILENAME="ClientApp.dppxp" />
                        <Path>ClientApp.exe</Path>
                        <Arguments>/arg1 /agr2 /arg3</Arguments>
                        <WorkingDirectory>c:\temp</WorkingDirectory>
                        <ExcludeImages>
                                <Image>ClassLibrary1.dll</Image>
                                <Image>ClassLibrary2.dll</Image>
                        </ExcludeImages>
                </Process>
                <Service CollectData="false" Start="true" RestartIfRunning="true"
                RestartAtEndOfRun="true">
                        <AnalysisOptions NM_METHOD_GRANULARITY="0" SESSION_DIR=""
                        SESSION_FILENAME="" />
                        <Name>iisadmin</Name>
                        <Host>remotemachine</Host>
                </Service>
        </Targets>
</ProductConfiguration>
```

Figure 7-6. The XML Configuration File Specifies the Session Details.

For detailed information about using the configuration file to manage data collection, see the online help.

QA engineers scan the session log file the following morning. If performance numbers have deteriorated, QA sends the session log to the appropriate developers. In this way, QA tracks the health of the application throughout the development cycle. If a problem appears, the development team has the session log file to use in quickly determining the nature of the problem. In addition, the development team knows that the problem was caused by a code change from the previous day, greatly reducing the amount of code it has to review to fix the problem.

Collecting Data from Distributed Applications

DevPartner can collect Performance Expert data from distributed application components that run on remote systems, provided the remote systems are properly licensed for remote data collection. Before you launch a remote session, be aware that a Performance Expert session

monitors a single process per run when run from Visual Studio or with `DPAnalysis.exe` from the command line using traditional command line syntax. The XML configuration file allows you to target more than one process or service in a single run of the application; however, it is usually best to target a single process in a Performance Expert session. If your application runs in multiple processes, simply rerun the application targeting the second process. Driving the application with a script or batch file ensures that you exercise the application identically in both sessions. For an overview, see "Automating Data Collection" on page 165. For detailed information, see the Performance Expert online help.

If necessary, you can collect the data (in a separate session file) for the second process or service in a single run of the application if you use `DPAnalysis.exe` with the XML configuration file option. Although you can collect data from two or more processes or services simultaneously, be aware that data collection overhead for multiple processes can affect interaction of the processes, as well as slowing the applications and inflating elapsed time values. If you collect Performance Expert data for multiple processes simultaneously, large timing values for disk I/O, network I/O, or synchronization wait time may reflect inflation by profiling overhead. Rerun the session targeting a single process to confirm that the timing values are large enough to merit investigation.

Performance Expert in the Development Cycle

Use DevPartner Performance Expert throughout the software development cycle. Software designers can use Performance Expert during the design and prototype phase to improve the speed and efficiency of their code. As the design progresses, regular testing helps to ensure that the prototype code meets minimal performance requirements. When the prototype is handed off to the development team, developers can feel comfortable reusing sections of the prototype, knowing that it has been tested for several critical performance issues.

Software developers should use Performance Expert frequently during development. Consider running Performance Expert in addition to unit tests prior to code check-ins. Just as the unit tests ensure that the component does what it is supposed to do without breaking other components, Performance Expert provides early warning of potential performance issues before the component is fully integrated into the application and therefore more difficult to fix.

Quality Assurance engineers can use Performance Expert as part of an automated test suite. Then designated engineers can review critical metrics in the session log files on a daily basis. If the session log suggests

a problem, the QA engineer can send the log file to the responsible developer so the problem can be addressed immediately.

Thus, all members of the software development team can benefit from running Performance Expert, from the design phase to final quality assurance testing. There is even a benefit for product management. At each critical milestone, Performance Expert session logs, coupled with before-and-after performance analysis session files, can be used to document that the product meets performance expectations.

Chapter 8
System Comparison
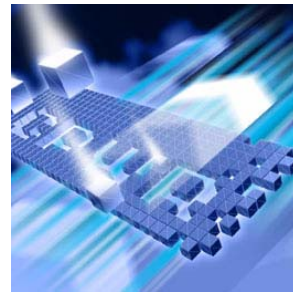
◆ What is the System Comparison Utility?

◆ Scenarios for Use

◆ Categories Compared

◆ Comparing Snapshots

◆ The System Comparison Service

◆ Running the Comparison Utility from the Command Line

◆ Software Development Kit

What is the System Comparison Utility?

The DevPartner System Comparison utility compares two computer systems, or compares two states of the same computer system, to identify differences. For example, if a service is running on one system, but stopped on another, System Comparison identifies that difference. System Comparison identifies differences in system files, installed products, drivers, and many other system characteristics.

The utility consists of a service, which takes nightly snapshots of a system, and the user interface, which enables you to compare snapshots to find differences. The utility also includes a command line interface and a software development kit.

Figure 8-1 illustrates the System Comparison user interface.
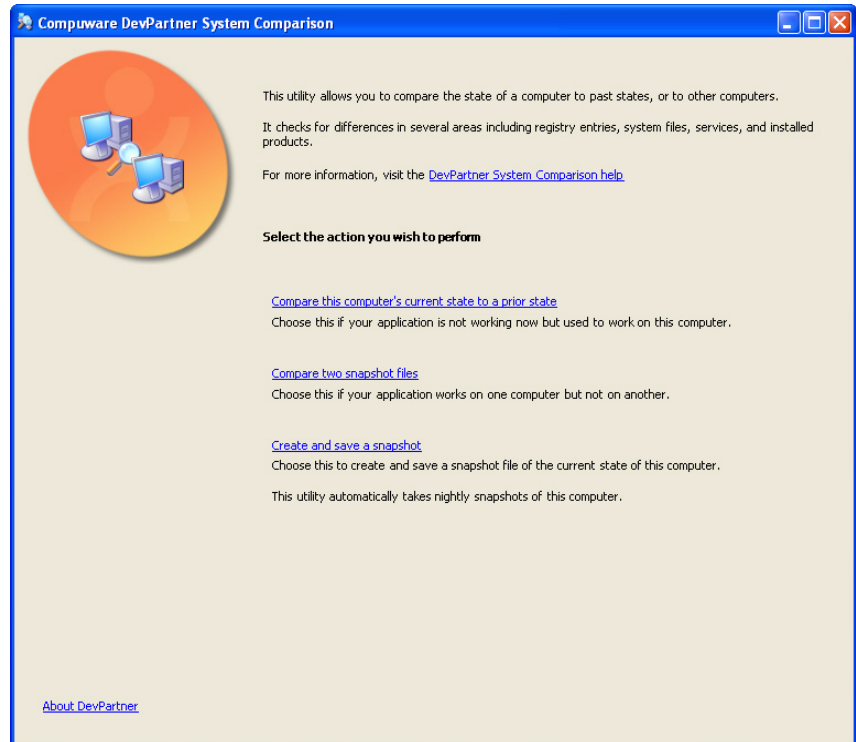
Figure 8-1. System Comparison User Interface

Scenarios for Use

The following are some general scenarios in which the System Comparison utility is used.

◆ To check how installation or removal of a product impacts computer services or settings

◆ To determine if system changes may have caused a product to stop working on a system on which it previously worked

◆ To determine why a product does not work, or works differently, on a certain system

◆ To determine the extent of the impact that changes to a product will make (for example, any impact on automated tests)

◆ To troubleshoot a product after it has been deployed to an end-user site

◆ Tp check that a new development system has all of the tools that were available on a previous development system

Categories Compared

When taking a snapshot, the System Comparison utility records the existence, version, and status of the items listed in the following table.

You can add additional categories to customize data acquisition by writing a System Comparison Plug-in, as described in "Software Development Kit" on page 177.

Table 8-1. Categories of Differences

Category	Differences Detected
System Info	Operating system .NET Framework Global Assembly Cache The Java Runtime System Environment variables
System Files	Operating system files in Windows\System32 Windows File Protection Cache in Windows\System32\dllcache - This folder contains operating system files that are used to maintain Windows if an operating system file is damaged. If a file is damaged or missing, it is automatically replaced from this folder without any intervention. Side-by-side assemblies in Windows\WinSxS
Installed Products	The products installed on the two systems. If the version number is detected, it is shown. The information is read from the Add/Remove Programs section of the registry.
Services	Differences in the installed services: • Service status (Running, Stopped, etc.) • Account used by the service • Service type • Services depended on

Table 8-1. Categories of Differences

Category	Differences Detected
Startup Items	Startup differences. This information is read from the following: • The Win.ini file found in the Windows directory. • The following registry key: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run • If possible, version information is included from the program file.
IE/Outlook Components	Internet Explorer and Outlook differences: Active Setup shows updated or missing Outlook / Internet Explorer components extracted from the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Active Setup\Installed Components Browser Helper objects extracted from the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects MIME mappings (mapping between MIME type and which application handles the MIME) extracted from the registry key: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Active Setup\MimeFeature objects. Internet Explorer extensions extracted from the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\Extensions
IIS	Differences in your Microsoft IIS installation. Differences in all installed Web applications and their settings, including: • Web server differences • SMTP server differences • FTP server differences
SQL Server	Differences in your Microsoft SQL installation: • Microsoft SQL settings from the registry. • Data about Microsoft SQL services and related services. • Settings in the `syscurconfigs` and `sysconfigures` tables in the master database for all installed instances. The System Comparison utility attempts to connect to SQL Server using integrated security. If SQL Server is not running, differences in the master database will not be collected. Note: For master database differences to be collected, the account under which you are running must have sufficient privilege to access these two tables.

Table 8-1. Categories of Differences

Category	Differences Detected
Drivers	Differences of all Drivers found: • Installed drivers • Status of drivers
Registry	Differences in specific sections of the registry. By default, no registry sections are collected, but differences in the following registry sections can be collected: HKEY_CLASSES_ROOT HKEY_LOCAL_MACHINE You can customize the sections of the registry to collect by editing the `RegistrySections.xml` file, found in the `Program Files\Compuware\DevPartner Studio\Comparison Tool\data` directory. You must have sufficient privilege to collect registry key data.
Files	Differences in the contents of directories and file properties from specific paths. By default, no files are included in the collection. You can customize the paths to collect by editing the `FileSections.xml` file, found in the `Program Files\Compuware\DevPartner Studio\Comparison Tool\data` directory.

Comparing Snapshots

When System Comparison compares two snapshots, it displays the differences between them as shown in Figure 8-2.

The comparison lists differences only; it does not list items that are the same in both snapshots. It reports components with different version numbers as different components.
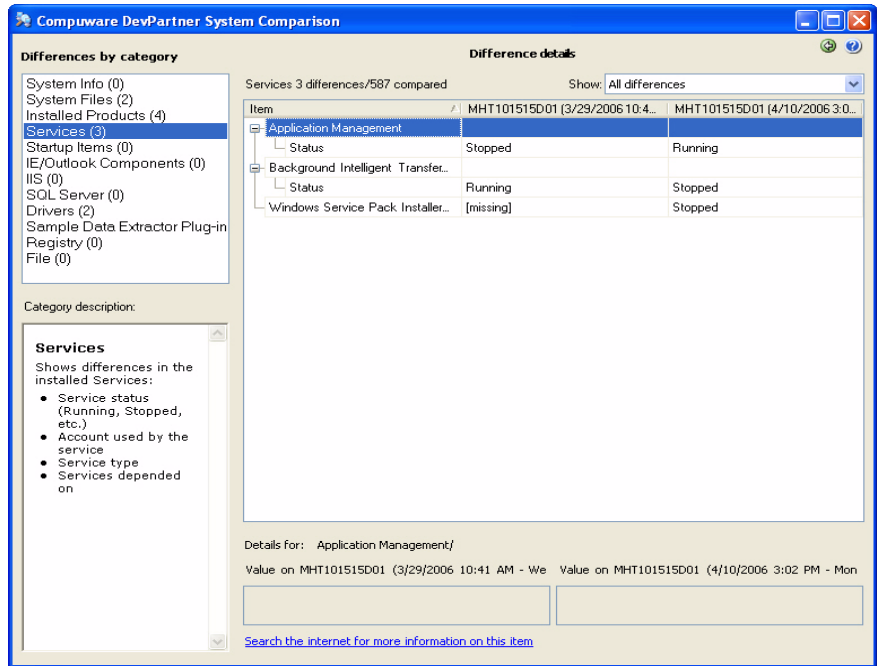
Figure 8-2. Comparing Snapshots

The Category pane on the left lists the categories of differences, with the number of differences for each category displayed in parentheses. Clicking on a category displays that category's differences in the Difference Details pane.

Near the bottom of the screen is a link to the internet. Clicking the link launches an internet search for items releated to the currently selected difference.

The System Comparison Service

The System Comparison service automatically takes a snapshot of the state of your machine at 2:10 a.m. daily if the machine is running. If the machine is powered off, it takes the snapshot five minutes after the next start-up.

The nightly snapshot service collects 21 snapshots, then begins deleting the oldest ones. You can change the number of kept snapshots by modifying the value in the System Comparison utility's settings file (`Compuware.Diff.Settings.xml`, located in the `Program Files\Compuware\DevPartner Studio\Comparison Tool\bin`

directory). The size of snapshot files varies depending on the amount of data collected. A typical file size is less than one megabyte.

The System Comparison service runs at minimum priority, but it does consume some system resources for several minutes while it runs. If you prefer, you can use the `Control Panel/Administrative Tools/ Services` screen to set the startup type to manual for the DevPartner System Comparison service, but you will lose the functionality of having snapshots created automatically.

Running the Comparison Utility from the Command Line

You can automate data collection and comparison using the two command line interfaces, `CommandLine.exe` and `CommandLineDiff.exe`.

◆ `CommandLine.exe` takes a snapshot of the current condition of your computer system. By default, it stores the snapshot in the last directory used to store snapshots, but you can specify an alternate directory as a parameter to the command line.

◆ `CommandLineDiff.exe` compares the values in two existing snapshot files and writes the resulting differences to an output file. Required parameters are 'computers' (it is a placeholder) and the names of the files to be compared. Optionally, you can specify the directory in which the output file will be written.

The output file is an XML file that can be read programmatically to check the results of the comparison. You cannot open this output file with the System Comparison utility's user interface.

The command line programs are located in the System Comparison utility's `\bin` directory (`\Program Files\Compuware\DevPartner Studio\Comparison Tool\bin` by default).

Software Development Kit

The System Comparison utility includes a Software Development Kit (SDK) that provides functionality for software developers, including:

◆ The ability to write a System Comparison Plug-in to specify additional information to be gathered during a snapshot.

The eleven categories of information gathered by the System Comparison utility (system information, system files, installed

products, and so on) are sufficient for most comparisons. If you require additional information to adequately compare systems, you can customize the System Comparison utility by writing a data acquisition plug-in.

For information about creating and testing plug-ins, refer to the file `WritingPlugin.html` in the `\sdk\Plugin` sub-directory in the System Comparison installation directory.

◆ The ability to use the Snapshot Application Program Interface (API) to embed function calls in an application to trigger a snapshot after the application is deployed.

The Snapshot API enables an application developer to control snapshot capability from within a deployed application. Should problems occur after the application is deployed, embedded API calls can trigger a snapshot that can assist with diagnosing the problem.

For information about the Snapshot API, refer to the file `SnapshotAPI.html` in the `\sdk\Snapshot` sub-directory in the System Comparison installation directory, and to the sample application `SampleSnapshotAPI.cs` in the `\sdk\Snapshot API` sub-directory in the System Comparison installation directory.

About DevPartner Studio Enterprise Edition and TrackRecord
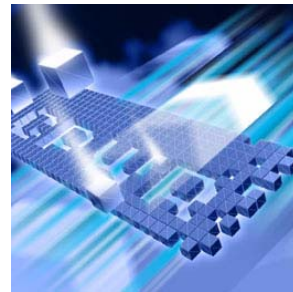
- ◆ What Is DevPartner Studio Enterprise Edition?
- ◆ The DevPartner Studio EE Solution
- ◆ Feature Overview
- ◆ About TrackRecord and DevPartner Studio
- ◆ DevPartner Studio Interaction with TrackRecord
- ◆ TrackRecord and DevPartner Studio Coverage Analysis

What Is DevPartner Studio Enterprise Edition?

DevPartner Studio Enterprise Edition (EE) can increase a manager's ability to predict when projects will reach a goal, such as a specified quality level or a deployment status. It gives project managers the concrete project details they need to keep software projects on schedule, and development team members the tools they need to accomplish their goals.

DevPartner Studio EE combines the features of several existing software solutions, and integrates them to provide a new class of functionality. In addition to the DevPartner features described in this manual, the Enterprise Edition also includes the following components:

| TrackRecord | Advanced change request management, task management, and workflow support for development teams |
| Reconcile | Practical requirements management for software development teams |

DevPartner Studio EE provides multiple ways to capture, manipulate, view, and track project data, including:

- ◆ Milestone-related summaries that provide a way to interpret and understand critical-path project data
- ◆ Customizable work flow for tracking data in a way that fits a company's development process
- ◆ Remote access to project information via a World Wide Web interface
- ◆ E-mail notification of changes to crucial project information

The Development Process

Each software development group establishes its own process, which is the set of steps that the group uses to get from the idea and design stage of a project to the implementation and delivery stage. DevPartner Studio EE fits in with a team's current process, and provides features to assist the fine-tuning of internal development procedures.

Examples of process include:

- ◆ Written requirements
- ◆ Systematic change control
- ◆ Technical reviews
- ◆ Quality assurance planning
- ◆ Implementation planning
- ◆ Automated source code control
- ◆ Estimation updates at major milestones

Projects that use no process often suffer from:

- ◆ Application redesigns and rewrites during testing
- ◆ Integration problems
- ◆ Defect corrections late in the development life cycle at great cost
- ◆ Expansion of requirements—a malady often called "thrashing"

Projects that use a well-managed process display a high degree of certainty about the status of the project in relation to its plan. Process also improves development team morale. In one 50-company survey, 60% of developers who rated morale as good or excellent worked for firms that emphasized process, as compared to 20% whose firms were the least process oriented.

DevPartner Studio EE adapts to a company's existing software development process, and provides tools to help teams enhance that process if they so choose. It provides a way to formalize a team's work flow, to make people answerable to that work flow, and to audit the entire process. Combining customizable work flow with automatic error detection improves software quality and streamlines the development process.

The DevPartner Studio EE Solution

DevPartner Studio EE provides solutions to problems commonly facing software development teams:

◆ Improved project control
◆ Higher software quality through enhanced code reliability
◆ Improved productivity

Improved Project Control

Keeping projects under control involves the ability to determine easily:

◆ What tasks has the team completed?
◆ What tasks remain uncompleted?
◆ How volatile is the application's code?
◆ How thoroughly tested is the application?
◆ How reliable is the application?

Dynamic Tracking of Project Information

DevPartner Studio EE excels at tracking dynamic project information using TrackRecord.

Numerous tools exist to plan software projects. These tools help determine resource allocation, schedules, critical-path tasks, and other vital information. Before DevPartner Studio EE, approved project plans became static data points. During real projects, schedules slip, programmers get pulled off projects to deal with escalated problems in other projects, and delivery conditions change. Project planning tools alone cannot easily help to deal with changing conditions, but the DevPartner Studio EE connection between Microsoft Project and TrackRecord allows dynamic recalculation of schedules.

Higher Software Quality

Developers and testing engineers will use DevPartner throughout the software development cycle.

Finding Problems Before Your Users Do

DevPartner Studio EE differs from other software project enhancement tools by encouraging a proactive and systematic approach to finding and fixing program anomalies, from outright bugs to code bottlenecks. The early location of a program's problems contributes to high quality in the final product. DevPartner Studio EE's debugging features assist developers, individually and collectively, throughout the development

cycle. DevPartner saves time for programmers by making errors easier to find and repair, and easy report creation increases the likelihood that developers will enter defect and feature reports.

Finding errors is just the start of a process. Errors need to be discovered, recorded, reproduced, and assigned a priority for repair. TrackRecord streamlines much of this process, which frees developers to be more productive while guaranteeing that problems do not get lost or forgotten in a hectic schedule.

Improved Productivity

As project milestones—crucial dates such as code freeze and deployment dates—approach, dynamic displays of project data, such as number of defects outstanding, code volatility, and team-wide code coverage statistics, help everyone on the team assess their progress toward goals.

Every DevPartner Studio EE user can create a unique *view* of the information in a project database.

◆ Managers can get a big-picture view of a project, can track whether crucial tests are being run, and can control quality more tightly
◆ Developers can create lists of tasks needing immediate attention, rank tasks according to priority, perform error checks and performance tests on their code, and focus their daily activities
◆ Testers can track bugs and the status of known problems, merge data generated by coverage runs, execute test plans, and organize daily activity
◆ Writers can track when specifications get published, when features get implemented, and when user interface changes get made
◆ Support coordinators can quickly locate information, such as known defects and configurations tested, to help customers resolve problems

In this way, individual users will be more productive by quickly retrieving just the information they need. Views such as the Milestone Summary provide a context for the display of this information.

Every software development project is different, and every company has different needs. Different parts of a single company need different information about ongoing projects. DevPartner Studio EE satisfies these requirements by offering flexibility in the design of projects, particularly in the types of information that get tracked.

Although DevPartner Studio EE provides numerous pre-built views of database information, every DevPartner Studio EE user will have unique requirements for the storage and display of project information. DevPartner Studio EE provides the flexibility to allow companies to

customize reports, forms, workflow, projects, and information types to fit their needs.

Feature Overview

DevPartner Studio EE provides the tools for accumulating and sharing software development project information. DevPartner Studio EE provides a rich set of features to facilitate the process of keeping track of a project under development.

Requirements Management

The crucial first step in any application development project is capturing the right set of end-user requirements. Next, those requirements must be effectively communicated to the development and testing teams. Reconcile provides a way to capture, organize, and distribute project requirements.

Using the familiar Microsoft Word as its editor, Reconcile provides a way to gather and refine requirements. Then, development teams can use the Reconcile Project Explorer to navigate requirement relationships. Reconcile requirements can be synchronized with QADirector to automatically create test procedures, and to correlate test results with test plans.

Reconcile integration with TrackRecord makes possible the association of defects and issues with project requirements. In this way, Reconcile and TrackRecord allow every development team member to stay up-to-date on project objectives.

Merging Coverage Data

The DevPartner Studio EE coverage feature generates information about the amount of a component's code that has been exercised or tested. Since different developers will likely work on different components, this individual coverage data will not tell a complete story about an application project. Each developer's local coverage report may need to be merged with other coverage results to obtain a complete picture of how much of the total project's code has been exercised.

DevPartner Studio EE allows the merging of sets of coverage data based on builds, configurations, users, or other criteria.

Project Activity Tracking

Tracking the various tasks and components of a software project helps to deal with the problem of complexity. As team members work on a specific task, new tasks needing attention at a future date often emerge. DevPartner Studio EE provides a way to record and track those tasks so that they will not get lost. Combining the work of individual developers requires attention to detail, coordination, and accurate recording of problems that will require consideration at a later date.

Tracking the level of testing being done, the number of faults being discovered, and the amount of coverage activity taking place can help a project manager anticipate and avoid problems. Two-way communication between DevPartner Studio EE and Microsoft Project can even automate schedule changes.

Automatic Notification of Changes

Timeliness promotes productivity. For example, prompt notification of:

◆ Newly-found high priority bugs helps managers reallocate resources to deal with shifting task priorities
◆ Newly-assigned tasks helps developers schedule their time more efficiently

While dynamic Outline reports provide the primary method for notifying users about changes to project data, the DevPartner Studio EE AutoAlert feature provides another way to notify one or more individuals when a tracked event occurs. AutoAlert lets you define flexible criteria for notifying remote or infrequent users of changes that might be of interest to them.

Each user who receives automatic mail notification sets up the notification criteria by creating special DevPartner Studio EE mail queries. AutoAlert monitors the DevPartner Studio EE database, periodically checking to see if any new items match the mail queries.

Whenever an item is entered or changed so that it matches one of the mail queries, DevPartner Studio EE automatically sends an e-mail message to the owner of the e-mail query. By using the TrackRecord software's flexible query engine, AutoAlert makes it possible for you to receive mail notification based on almost any criteria.

Customizable Workflow

Every software development team needs a way to make sure that certain tasks get completed, often in a specific order. Quality Assurance cannot test a repaired defect, for example, until the fix gets logged as integrated

with the rest of the application under development. DevPartner Studio EE allows, but does not require, setting up a workflow that works in this manner.

DevPartner Studio EE provides a mechanism to implement an ordered workflow. Teams can design this workflow to restrict who can move an item of project data from one stage in the workflow life cycle to another. The workflow and its enforcement policies can require certain information under specified conditions. These policies provide a way to make team members accountable to the process the project uses.

Remote Access via the Web

When members of a development team work at remote locations, they can still have access to project data. The DevPartner Studio EE WebServer provides remote access via standard Web connections to allow users to view, enter, and change crucial project data.

Central Store of Shared Information

DevPartner Studio EE provides a robust client-server-based repository for sharing information. This repository uses an object-oriented database that is programmatically accessible through ActiveX (formerly OLE automation) interfaces. The repository provides the underlying infrastructure to enable groups to work together while each member works separately.

An extensible and flexible database structure, based on information types, forms the core of DevPartner Studio EE's repository, and provides its power.

About TrackRecord and DevPartner Studio

TrackRecord is part of the DevPartner Studio Enterprise Edition suite of software development tools. These applications automatically generate and store information about the detection, diagnosis, and resolution of software problems.

Developers can use TrackRecord to capture this information, along with other project information, such as milestone dates, to help resolve problems quickly and consistently.

Note: Integration of TrackRecord and DevPartner Studio is version depen-
dent. You might need to upgrade your DevPartner Studio or Track-
Record software if you purchased the tools at separate times.

DevPartner Studio Interaction with TrackRecord

DevPartner Studio provides toolbar buttons and menu selections that allow the submission of defects to TrackRecord databases.

DevPartner Toolbar Buttons

The DevPartner toolbar buttons let you enter DevPartner Defects. Clicking these buttons opens a defect form, allowing you to key information into the DevPartner Studio database.

Defect Submissions

Submitting a DevPartner Studio Defect starts with highlighting an item from a DevPartner Studio debug display.

Entering a Defect from DevPartner

Complete the following steps to enter a defect from DevPartner:

1   Choose **Submit Defect** from a DevPartner menu or toolbar.

    Alternatively, choose a Submit Defect button from a DevPartner feature toolbar.

    TrackRecord opens either a blank Defect form, or a form with some fileds prefilled with relevant data.

2   Enter other needed information into the defect report.

3   Click **Save** and **Close**.

TrackRecord and DevPartner Studio Coverage Analysis

DevPartner coverage users can merge session files that accumulate within their private work space. These merged sessions indicate how much testing that developer's code has received over time.

With DevPartner Studio and TrackRecord, coverage sessions can be merged and filtered across users and environments. Merging coverage sessions from *all* the developers working on an application lets a manager or test coordinator determine how much of an application's total code base has been exercised by test programs.

Refer to the coverage documentation and online help for information about how to use DevPartner coverage.

Merging coverage sessions entails two steps: creating a coverage merge set, and merging the sessions. A developer typically chooses what coverage sessions should be merged and what sessions should be excluded. Criteria for identifying sessions to merge can include the following.

◆ Application component
◆ Date
◆ Memory
◆ Milestone
◆ Operating System
◆ Person
◆ Project

You can match one of these selections to a specific value, to any value, or to any value except one you specify.

Creating Criteria for Merge Coverage Operation

To create criteria for a merge coverage operation, complete the following steps:

1   In TrackRecord, select Merge Coverage Sessions from the Tools menu.

2   Select a target from the left-most list box.

3   Select a match criteria from the right-most list box.

4   If you selected "is equal to" in Step 2, select a value from the bottom list box.

    For example, if you selected "Operating System is equal to" in the two top lists, you would select a value, such as "Windows 98," from the lower list.

5   Click **Add**.

6   Click **Next** to view the sessions that met your criteria.

Merging Coverage Sessions

To merge coverage sessions, complete the following procedure:

1   Click the check box next to a session to toggle it on or off.

    When checked, that session will be merged with the other files selected. If unchecked, that session will not be merged with the other selected files.

2   Click **Merge**.

The DevPartner coverage main dialog box opens and displays a bar graph and statistics about the amount of lines and functions exercised by your unit tests.

Index