Micro Focus Enterprise Analyzer 3.4

Creating Components

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

2013-04-26

Contents

Introducing Component Maker

The Component Maker tool includes the Dead Code Elimination slicing algorithm that lets you remove all of the dead code from a program. You can create a self-contained program, called a component from the sliced code or simply generate a HyperView list of sliced constructs for further analysis. You can mark and colorize the constructs in the HyperView Source pane.

Componentization Methods

The supported componentization methods slice logic not only from program executables but associated include files as well. Dead Code Elimination is an optimization tool built into the main methods and offered separately in case you want to use it on a standalone basis.

Note: Component Maker does not follow CALL statements into other programs to determine whether passed data items are actually modified by those programs. It makes the conservative assumption that all passed data items are modified. That guarantees that no dependencies are lost.

Dead Code Elimination (DCE)

Dead Code Elimination is an option in each of the main component extraction methods, but you can also perform it on a standalone basis. For each program analyzed for dead code, standalone DCE generates a component that consists of the original source code minus any unreferenced data items or unreachable procedural statements.

Note: Use the batch DCE feature to find dead code across your project. If you are licensed to use the Batch Refresh Process (BRP), you can use it to perform dead code elimination across a workspace.

Language Support

The following table describes the extraction methods available for Component Maker-supported languages.

Method	COBOL	PL/I	Natural	RPG
Dead Code Elimination	Yes	Yes	Yes	Yes

Componentization Outputs

The first step in the componentization process, called *extraction*, generates the following outputs:

- The source file that comprises the component.
- An abstract repository object, or *logical component*, that gives you access to the source file in Enterprise Analyzer.
- A HyperView list of sliced constructs, which you can mark and colorize in the HyperView Source pane.

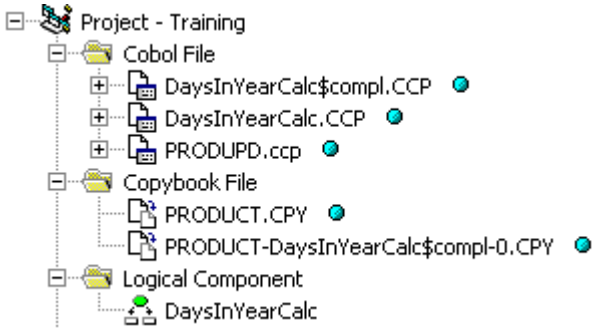
Note: Sliced data declarations are not marked and colorized.

The second step, called *conversion*, registers the source files in your repository, creating repository objects for the generated components and their corresponding copybooks.

Component Maker lets you execute the extraction and conversion steps independently or in combination, depending on your needs:

- If you want to analyze the components further, transform them, or even generate components from them, you will want to register the component source files in your repository and verify them, just as you would register and verify a source file from the original legacy application.
- If you are interested only in deploying the components in your production environment, you can skip the conversion step and avoid cluttering your repository.

The figure below shows how the componentization outputs are represented in the Repository Browser after conversion and verification of a Cobol component called DaysInYearCalc. PRODUPD is the program the component was extracted from.



Component Maker Basics

Component Maker is a Interactive Analysis-based tool that you can invoke from within Interactive Analysis itself:

- Start the tool in Interactive Analysis by selecting the program you want to slice in the Enterprise Analyzer Repository Browser and choosing **Analyze > Interactive Analysis**. In the Interactive Analysis window, choose **View > Components**.

  Note: Choose **View > Logic Analyzer** if you are using Logic Analyzer.

The Components pane consists of a hierarchy of views that let you specify the logical components you want to manipulate:

- The Types view lists the types of logical components you can create.
- The List view displays logical components of the selected type.
- The Details view displays the details for the selected logical component in two tabs, Properties and Components. The Properties tab displays extraction properties for the logical component. The Components tab lists the files generated for the logical component.

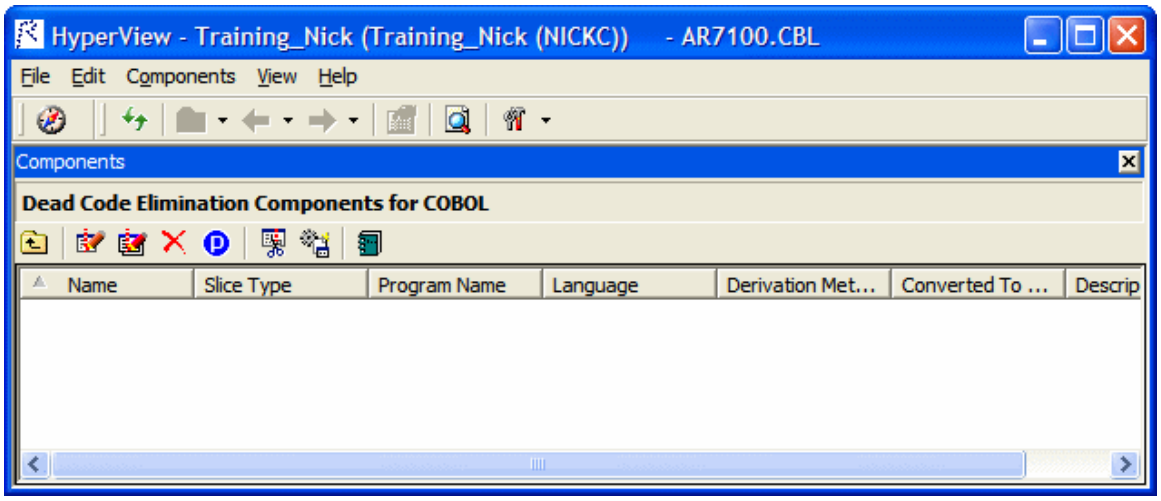
Getting Started in the Components Pane

You do most of your work in Component Maker in the Components pane. To illustrate how you extract a logical component in the Components pane, let's look at the simplest task you can perform in Component Maker, Dead Code Elimination (DCE).

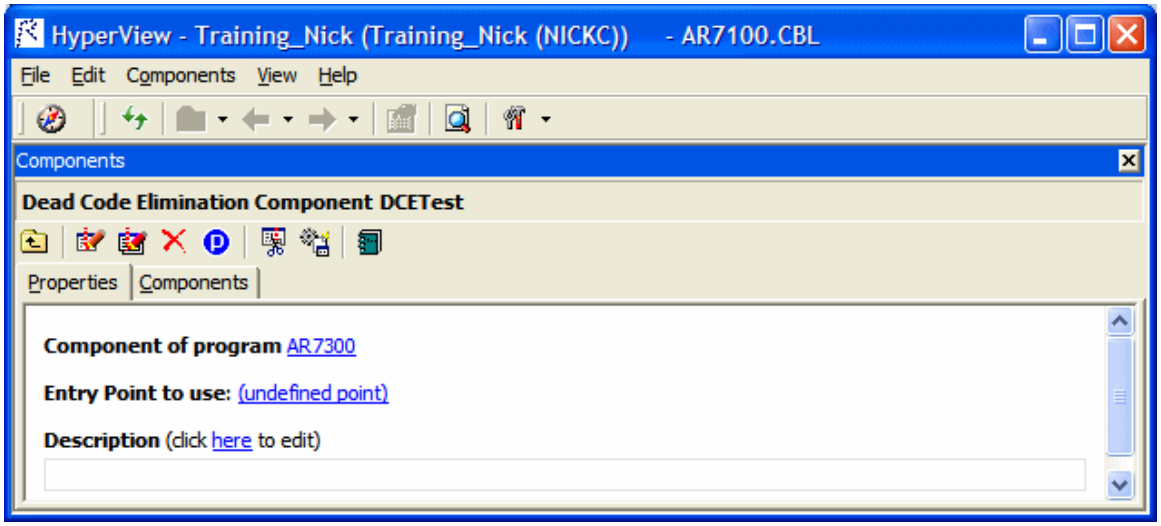
Note: The following exercise deliberately avoids describing the properties and options you can set for DCE. See the relevant help topics for details.

1. In the Components pane, double-click **Dead Code Elimination**. The view shown in the figure below opens. This view shows the DCE-based logical components created for the programs in the current project.

   Tip: Click the button on the tool bar to restrict the display to logical components created for the selected program.



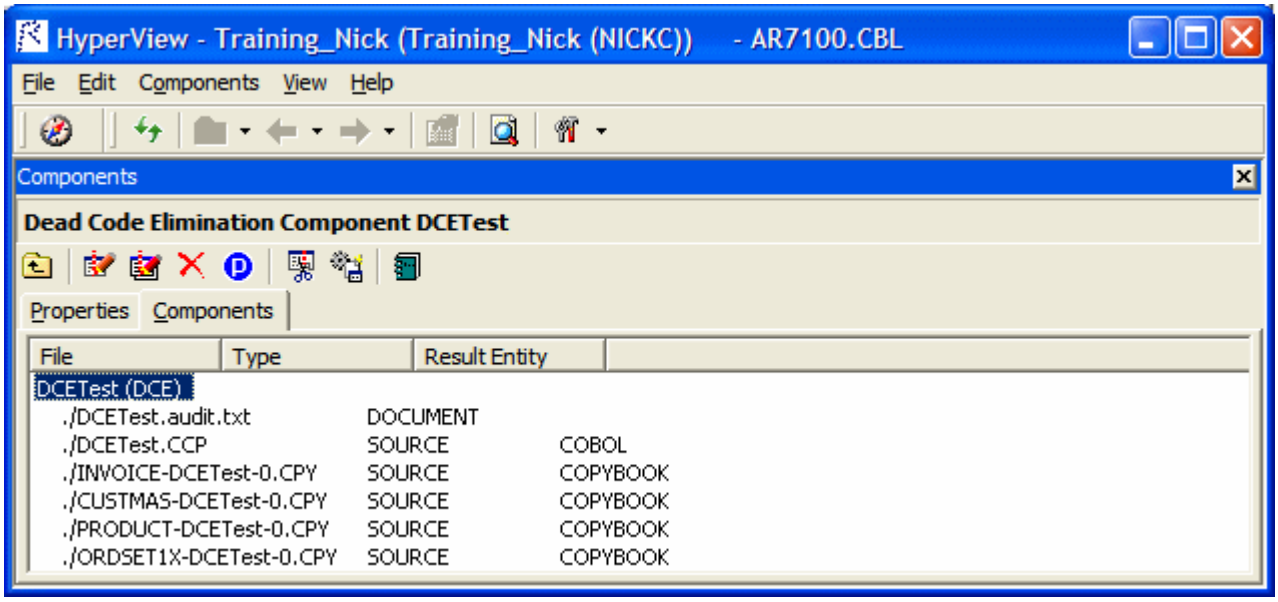
2. Select the program you want to analyze for dead code in the HyperView Objects pane and click the button. To analyze the entire project of which the program is a part, click the button.
3. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new components to the list of components. If you selected batch mode, Component Maker creates a logical component for each program in the project, appending *_n* to the name of the component.
4. Double-click a component to edit its properties. The view shown in the figure below opens. The **Component of program** field contains the name of the selected program.



5. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.

Note: This field is shown only for Cobol programs.

6. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
7. Click the button on the tool bar to navigate to the list of components, then repeat the procedure for each component you want to extract.
8. In the list of components, select each component you want to extract and click the button on the tool bar. You are prompted to confirm that you want to extract the components. Click **OK**.
9. The Extraction Options dialog opens. Set extraction options as described in the relevant help topic. When you are satisfied with your choices, click **Finish**.
10. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** to view the errors or warnings in the Activity Log. Otherwise, click **No**.
11. Assuming the extraction executed without errors, the view shown in the figure below opens. Click the Components tab to display a list of the component source files that were generated for the logical component and an audit report if you requested one. Click an item in the list to view the read-only text for the item.



Creating Components

To create a component, select the program you want to slice in the HyperView Objects pane. In the Types view, select the type of logical component you want to create and click the button on the tool bar. (You can also click the button in the List or Details view.) A dialog opens where you can enter the name of the new component in the text field. Click **OK**.

Extracting Components

To extract a single logical component, select the component you want to extract in the List view and click the button on the tool bar. To extract multiple logical components, select the type of the components

you want to extract in the Types view and click the  button. You are prompted to confirm that you want to continue. Click **OK**.

Tip: Logical components are converted as well as extracted if the **Convert Resulting Components to Legacy Objects** is set in the Component Conversion Options pane.

Converting Components

To convert a single logical component, select the component you want to convert in the List view and click the  button on the tool bar. To convert multiple logical components, select the type of the components you want to convert in the Types view and click the  button. You are prompted to confirm that you want to continue. Click **OK**.

Deleting Components

To delete a logical component, select it in the List view and click the  button on the tool bar.

Note: Deleting a logical component does not delete the component and copybook repository objects. You must delete these objects manually in the Repository Browser.

Viewing the Text for Generated Files

To view the read-only text for a generated file, click the file in the list of generated files for in the Components tab.

Tip: You can also view the text for a generated file in the Enterprise Analyzer main window. In the Repository Browser Logical Component folder, click the component whose generated files you want to view.

Restricting the Display to Program-Related Components

To restrict the display to logical components of a given program, select the program and click the  button on the tool bar. The button is a toggle. Click it again to revert to the generic display.

Working with HyperView Lists

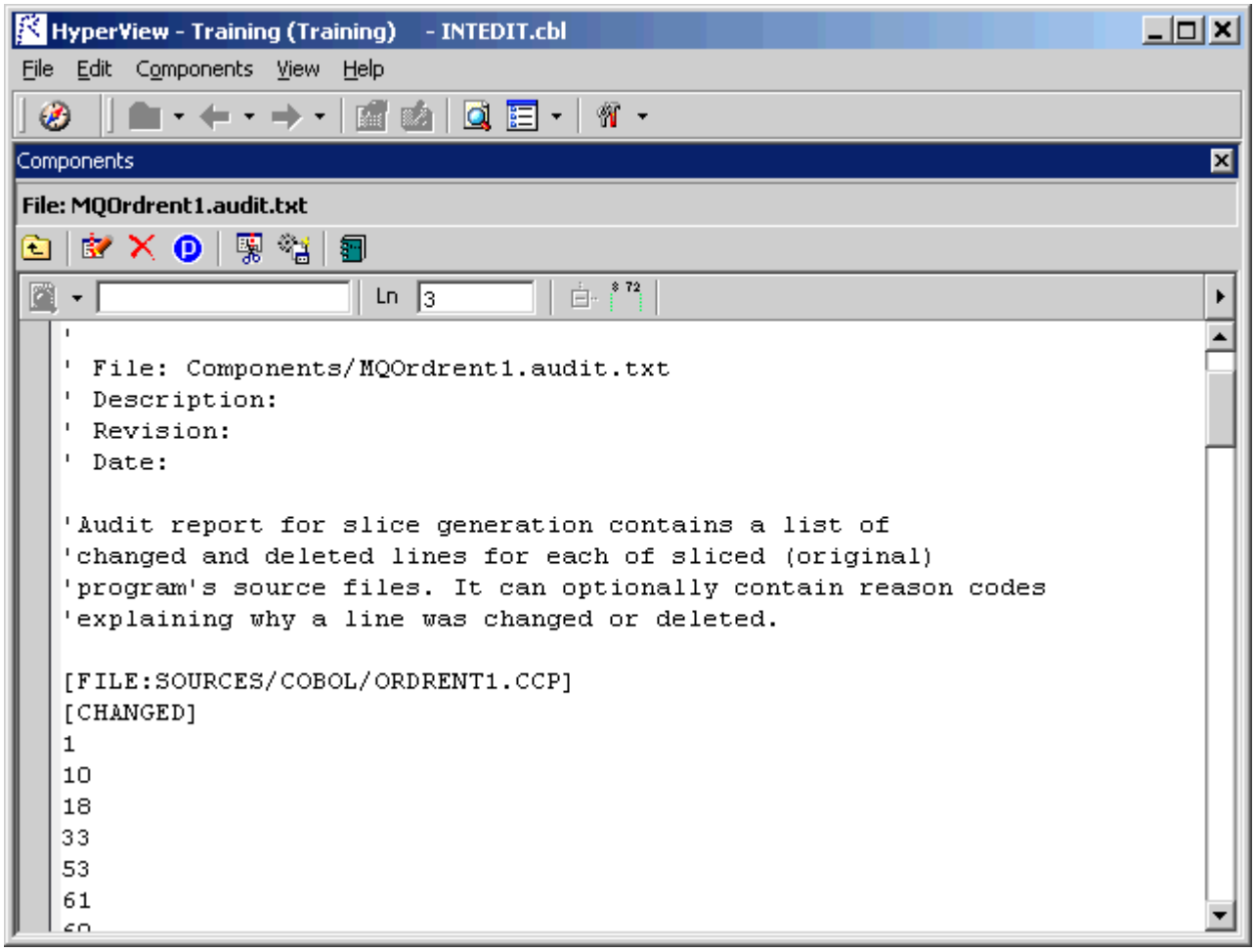
When you extract a logical component, Component Maker generates a HyperView list of sliced constructs. The list has the same name as the component. You can view the list in the Logic Analyzer folder in Clipper.

To mark and colorize sliced constructs in the list, select the list in Clipper and click the  button on the tool bar. To mark and colorize sliced constructs in a single file, select the file in the List view and click the  button. To mark and colorize a single construct, select it in the File view and click the  button. Click the  button again to turn off marking and colorizing.

Viewing Audit Reports

An *audit report* contains a list of changed and deleted lines in the source files (including copybooks) from which a logical component was extracted. The report has a name of the form <*component*>.audit.txt. Click the report in the Components tab to view its text.

An audit report optionally includes reason codes explaining why a line was changed or deleted. A reason code is a number keyed to the explanation for a change (for example, reason code 12 for computation-based componentization is RemoveUnusedVALUEs).



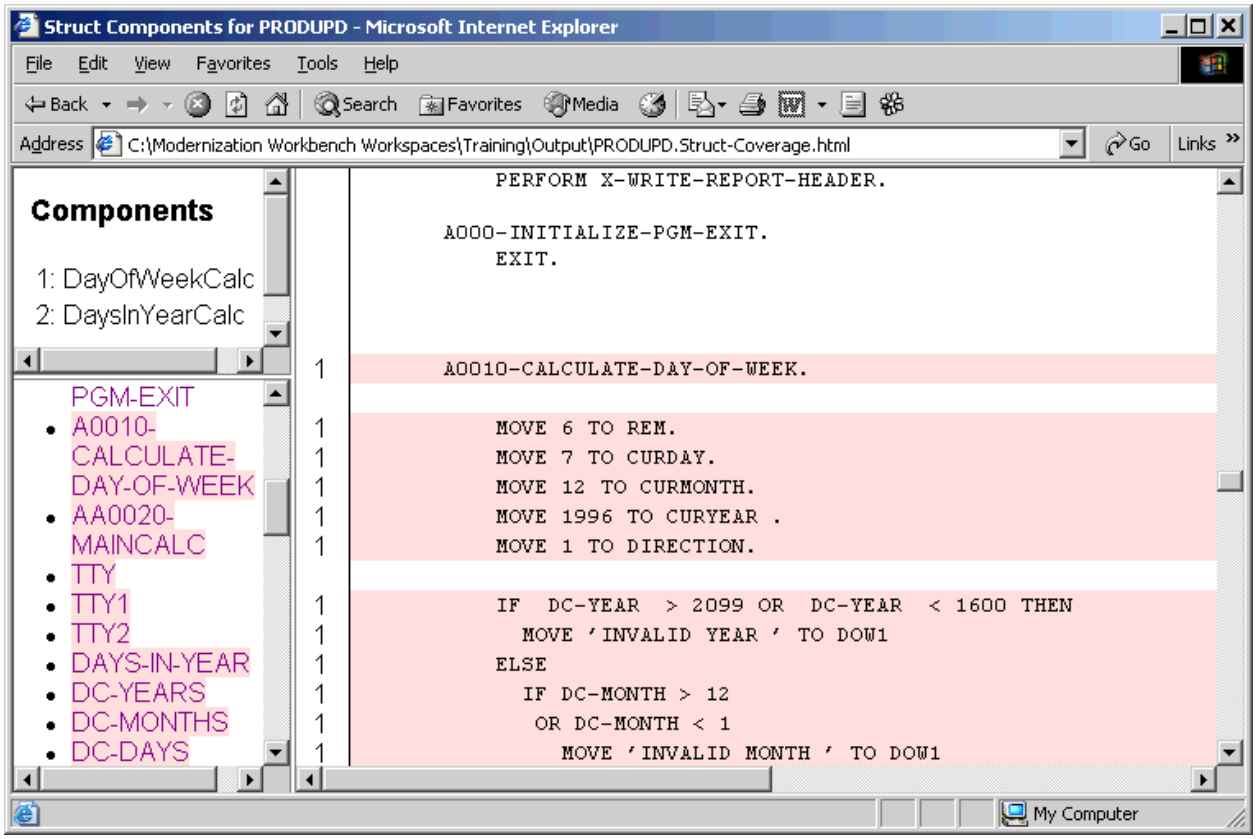
Generating Coverage Reports

A *coverage report* shows the extent to which a source program has been "componentized":

- The top-left pane lists each component of a given type extracted from the program.
- The bottom-left pane lists the paragraphs in the program. Click on a paragraph to navigate to it in the righthand pane.
- The righthand pane displays the text of the program with extracted code shaded in pink. The numbers to the left of the extracted code identify the component to which it was extracted.

To generate coverage reports, click  on the Component Maker tool bar. The reports are listed in the Generated Document folder in the Repository Browser. Report names are of the form <*program*>-<*method*>-Coverage. Double-click a report to view it in a Web browser.

 **Note:** Reports are created for each program in the current project.


Struct Components for PRODUPD - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\Modernization Workbench Workspaces\Training\Output\PRODUPD.Struct-Coverage.html
Go Links
Components
1: DayOfWeekCalc
2: DaysInYearCalc
PGM-EXIT
A0010-CALCULATE-DAY-OF-WEEK
AA0020-MAINCALC
TTY
TTY1
TTY2
DAYS-IN-YEAR
DC-YEARS
DC-MONTHS
DC-DAYS
PERFORM X-WRITE-REPORT-HEADER.
A000-INITIALIZE-PGM-EXIT.
EXIT.
1 A0010-CALCULATE-DAY-OF-WEEK.
1 MOVE 6 TO REM.
1 MOVE 7 TO CURDAY.
1 MOVE 12 TO CURMONTH.
1 MOVE 1996 TO CURYEAR .
1 MOVE 1 TO DIRECTION.
1 IF DC-YEAR > 2099 OR DC-YEAR < 1600 THEN
1 MOVE 'INVALID YEAR ' TO DOW1
1 ELSE
1 IF DC-MONTH > 12
1 OR DC-MONTH < 1
1 MOVE 'INVALID MONTH ' TO DOW1
My Computer

Setting Component Maker Options

It's a good idea to become familiar with the component extraction options before beginning your work in Component Maker. Each extraction method has a different set of options, and each set differs for the supported object types. Extraction options are project-based, so they apply to every program in the current Enterprise Analyzer project.

You can set Component Maker extraction options in the standard Project Options window or in the extraction options dialog that opens when you create a component. To open the standard Project Options window, choose **Options > Project Options**. In the Project Options window, click the **Component Maker** tab.

Setting General Options

The table below describes the Component Maker General extraction options.

Option	Language	Description
Add Program Name as Prefix	Cobol, Natural, PL/I, RPG	Prepend the name of the sliced program to the component name you specified when you created the component, in the form <*program*> $<*component*>.
Generate Slice	Cobol, Natural, PL/I, RPG	Generate both a HyperView list of sliced constructs and a component.
Keep Legacy Copybooks	Cobol, RPG	Do not generate modified copybooks for the component. Modified copybooks have names of the form <*copybook*>-<*component*>-*n*, where *n* is a number ensuring the uniqueness of the copybook name when multiple instances of a copybook are generated for the same component.  **Note:** Component Maker issues a warning if including the original copybooks in the component would result in an error.
Keep Legacy Includes	PL/I	Do not generate modified program include files for the component. The layout and commentary of the sliced program is preserved.
Keep Legacy Macros	PL/I	Do not expand macros for the component. The layout and commentary of the sliced program is preserved.
Preserve Legacy Includes	Natural	Do not generate modified program include files for the component.
Rename Program Entries	Cobol	Prepend the name of the component to inner entry points, in the form <*component*>-<*entrypoint*>. This ensures that entry point names are unique and that the Enterprise Analyzer parser can verify the component successfully. Unset this option if you need to preserve the original names of the inner entry points.

Setting Interface Options

The table below describes the Component Maker Interface extraction options.

Option	Language	Description
Blocking	Cobol	If you are performing a parameterized computation-based extraction and want to use blocking, click the **More** button. A dialog opens, where you can select the blocking option and the types of statements you want to block.  **Note:** Choose Use Blocking from Component Definitions if you want to block statements in a HyperView list.
Create CICS Program	Cobol	Create COMMAREAS for parameter exchange in generated slices.
Generate Parameterized Components	Cobol	Extract parameterized slices.

Setting Optimize Options

The table below describes the Component Maker Optimize extraction options.

Option	Language	Description
No changes	Cobol, Natural, RPG	Do not remove unused data items from the component.
Preserve Original Paragraphs	Cobol	Generate paragraph labels even for paragraphs that are not actually used in the source code (for example, empty paragraphs for which there are no PERFORMs).  **Note:** This option also affects refactoring. When the option is set, paragraphs in the same "basic block" are defragmented separately. Otherwise, they are defragmented as a unit.

Remove Redundant NEXT SENTENCE	Cobol	Remove NEXT SENTENCE clauses by changing the bodies of corresponding IF statements, such that:
		`IF A=1`
		`  NEXT SENTENCE`
		`ELSE`
		`  ...`
		`END-IF.`
		is generated as:
		`IF NOT (A=1)`
		`  ...`
		`END-IF.`

Remove/Replace Unused Fields with FILLERs	Cobol, Natural, RPG	Remove unused any-level structures and replace unused fields in a used structure with FILLERs. Set this option if removing a field completely from a

Option	Language	Description
		structure would adversely affect memory distribution.  **Note:** If you select Keep Legacy copybooks in the General component extraction options, Component Maker removes or replaces with FILLERs only unused inline data items.
Remove Unreachable Code	Cobol, RPG	Remove unreachable procedural statements.

Remove Unused Any-Level Structures	Cobol, Natural, RPG	Remove unused structures at any data level, if all their parents and children are unused. For the example below, D, E, F, and G are removed:
		`DEFINE DATA LOCAL`
		`1 #A`
		`  2 #B`
		`    3 #C`
		`  2 #D`
		`    3 #E`
		`    3 #F`
		`1 #G`

Remove Unused Level-1 Structures	Cobol, Natural, RPG	Remove only unused level-1 structures, and then only if all their children are unused. If, in the following example, only B is used, only G is removed:
		`DEFINE DATA LOCAL`
		`1 #A`
		`  2 #B`
		`    3 #C`
		`  2 #D`
		`    3 #E`
		`    3 #F`
		`1 #G`

Replace Section PERFORMs by Paragraph PERFORMs	Cobol	Replace PERFORM section statements by equivalent PERFORM paragraph statements.

Roll-Up Nested IFs	Cobol	Roll up embedded IF statements in the top-level IF statement, such that:
		`IF A=1`
		`    IF B=2`
		is generated as:
		`IF (A=1) AND (B=2)`

Setting Document Options

The table below describes the Component Maker Document extraction options.

Option	Language	Description
Comment-out Sliced-off Legacy Code	Cobol, RPG	Retain but comment out unused code in the component source. In the Comment Prefix field, enter descriptive text (up to six characters) for the commented-out lines.

Option	Language	Description
Emphasize Component/Include in Coverage Report	Cobol, Natural, PL/I, RPG	Generate a HyperView list of sliced constructs and colorize the constructs in the Coverage Report.
Generate Audit Report	Cobol	Generate an audit report.
Generate Support Comments	Cobol, RPG	Include comments in the component source that identify the component properties you specified, such as the starting and ending paragraphs for a structure-based Cobol component.
Include Reason Codes	Cobol	Include reason codes in the audit report explaining why a line was changed or deleted.  **Note:** Generating reason codes is very memory-intensive and may cause crashes for extractions from large programs.
List Options in Component Header and in Separate Document	Cobol, RPG	Include a list of extraction option settings in the component header and in a separate text file. The text file has a name of the form <*component*>.BRE.options.txt.
Mark Modified Legacy Code	Cobol, RPG	Mark modified code in the component source. In the Comment Prefix field, enter descriptive text (up to six characters) for the modified lines.
Print Calculated Values as Comments	Cobol	For domain-based component extraction only, print the calculated values of variables as comments. Alternatively, you can substitute the calculated values of variables for the variables themselves.
Use Left Column for Marks	Cobol, RPG	Place the descriptive text for commented-out or modified lines in the lefthand column of the line. Otherwise, the text appears in the righthand column.

Setting Component Type-Specific Options

Component type-specific extraction options determine how Component Maker performs tasks specific to each componentization method.

Setting Component Conversion Options

The table below describes the Component Maker Component Conversion extraction options.

Option	Language	Description
Convert Resulting Components	Cobol, Natural, PL/I, RPG	Convert as well as extract the logical component.
Keep Old Legacy Objects	Cobol, Natural, PL/I, RPG	Preserve existing repository objects for the converted component (copybooks, for example). If you select this option, delete the repository object for the component itself before performing the extraction, or the new component object will not be created.
Remove Components after Successful Conversion	Cobol, Natural, PL/I, RPG	Remove logical components from the current project after new component objects are created.

Option	Language	Description
Replace Old Legacy Objects	Cobol, Natural, PL/I, RPG	Replace existing repository objects for the converted component.  **Note:** This option controls conversion behavior even when you perform the conversion independently from the extraction. If you are converting a component independently and want to change this setting, select Convert Resulting Components to Legacy Objects, specify the behavior you want, and then deselect Convert Resulting Components to Legacy Objects.

Eliminating Dead Code

Dead Code Elimination (DCE) is an option in each of the main component extraction methods, but you can also perform it on a standalone basis. For each program analyzed for dead code, DCE generates a component that consists of the original source code minus any unreferenced data items or unreachable procedural statements. Optionally, you can have DCE comment out dead code in Cobol and Natural applications, rather than remove it.

Note: Use the batch DCE feature to find dead code across your project. If you are licensed to use the Batch Refresh Process (BRP), you can use it to perform dead code elimination across a workspace.

Generating Dead Code Statistics

Set the **Perform Dead Code Analysis** option in the project verification options if you want the parser to collect statistics on the number of unreachable statements and dead data items in a program, and to mark the constructs as dead in HyperView. You can view the statistics in the Legacy Estimation tool, as described in *Analyzing Projects* in the product documentation set.

Note: You do not need to set this option to perform dead code elimination in Component Maker.

For Cobol programs, you can use a DCE coverage report to identify dead code in a source program. The report displays the text of the source program with its "live," or extracted, code shaded in pink.

Understanding Dead Code Elimination

Let's look at a simple before-and-after example to see what you can expect from Dead Code Elimination.

Before:

```
WORKING-STORAGE SECTION.

 01 USED-VARS.
  05 USED1 PIC 9.

 01 DEAD-VARS.
  05 DEAD1 PIC 9.
  05 DEAD2 PIC X.

 PROCEDURE DIVISION.

 FIRST-USED-PARA.
  MOVE 1 TO USED1.
  GO TO SECOND-USED-PARA.
  MOVE 2 TO USED1.

 DEAD-PARA1.
  MOVE 0 TO DEAD2.

 SECOND-USED PARA.
  MOVE 3 TO USED1.
  STOP RUN.
```

After:

```
WORKING-STORAGE SECTION.

 01 USED-VARS.
  05 USED1 PIC 9.

 PROCEDURE DIVISION.

 FIRST-USED-PARA.
  MOVE 1 TO USED1.
  GO TO SECOND-USED-PARA.

 SECOND-USED PARA.
  MOVE 3 TO USED1.
  STOP RUN.
```

Extracting Optimized Components

Follow the instructions below to extract optimized components for all supported languages.

1. Select the program you want to analyze for dead code in the HyperView Objects pane and click the  button. To analyze the entire project of which the program is a part, click the  button.
2. A dialog opens where you can enter the name of the new component in the text field. Click **OK**. Component Maker adds the new components to the list of components. If you selected batch mode, Component Maker creates a logical component for each program in the project, appending *_n* to the name of the component.
3. In the **Entry Point to use** field, click the link for the current selection and choose the entry point you want to use in the pop-up menu. To unset an entry point, click it and choose **Unset** in the pop-up menu.
4. In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
5. Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
6. The Extraction Options dialog opens. Set options for the extraction and click **Finish.**
7. Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** in the notification dialog to view the errors or warnings in the Activity Log. Otherwise, click **No.**

Technical Details

This appendix gives technical details of Component Maker behavior for a handful of narrowly focused verification and extraction options; for Cobol parameterized slice generation; and for Cobol arithmetic exception handling.

Verification Options

This section describes how a number of verification options may affect component extraction. For more information on the verification options, see *Preparing Projects* in the product documentation set.

Use Special IMS Calling Conventions

Select **Use Special IMS Calling Conventions** in the project verification options if you want to show dependencies and analyze CALL 'CBLTDLI' statements for the CHNG value of their first parameter, and if the value of the third parameter is known, then generate Calls relationship in the repository.

For example:

```
MOVE 'CHNG' TO WS-IMS-FUNC-CODE
MOVE 'MGRW280' TO WS-IMS-TRANSACTION
CALL 'CBLTDLI' USING WS-IMS-FUNC-CODE
 LS03-ALT-MOD-PCB
 WS-IMS-TRANSACTION
```

When both WS-IMS-FUNC-CODE = 'CHNG' and WS-IMS-TRANSACTION have known values, the repository is populated with the CALL relationship between the current program and the WS-IMS-TRANSACTION <value> program (in the example, 'MGRW280').

Override CICS Program Terminations

Select **Override CICS Program Terminations** in the project verification options if you want the parser to interpret CICS RETURN, XCTL, and ABEND commands in Cobol files as not terminating program execution.

If the source program contains CICS HANDLE CONDITION handlers, for example, some exceptions can arise only on execution of CICS RETURN. For this reason, if you want to see the code of the corresponding handler in the component, you need to check the override box. Otherwise, the call of the handler and hence the handler's code are unreachable.

Support CICS HANDLE Statements

Select **Support CICS HANDLE statements** in the project verification options if you want the parser to recognize CICS HANDLE statements in Cobol files. EXEC CICS HANDLE statements require processing to detect all dependencies with error-handling statements. That may result in adding extra paragraphs to a component.

Perform Unisys TIP and DPS Calls Analysis

Select **Perform Unisys TIP and DPS Calls Analysis** in the project verification options if you are working on a project containing Unisys 2200 Cobol files and need to perform TIP and DPS calls analysis.

This analysis tries to determine the name (value of the data item of size 8 and offset 20 from the beginning of form-header) of the screen form used in input/output operation (at CALL 'D$READ', 'D$SEND', 'D $SENDF', 'D$SENDF1') and establish the repository relationships ProgramSendsMap and ProgramReadsMap between the program being analyzed and the detected screen.

For example:

```
01 SCREEN-946.
 02 SCREEN-946-HEADER.
 05 FILLER PIC X(2)VALUE SPACES.
 05 FILLER PIC 9(5)COMP VALUE ZERO.
 05 FILLER PIC X(4)VALUE SPACES.
 05 S946-FILLER PIC X(8) VALUE '$DPS$SWS'
 05 S946-NUMBER PIC 9(4) VALUE 946.
 05 S946-NAME PIC X(8) VALUE 'SCRN946'.
CALL 'D$READ USING DPS-STATUS, SCREEN-946.
```

Relationship ProgramSendsMap is established between the program and screen 'SCRN946'.

Note: Select **DPS routines may end with error** if you want to perform call analysis of DPS routines that end in an error.

Perform Unisys Common-Storage Analysis

Select **Perform Unisys Common-Storage Analysis** in the project verification options if you want the system to include in the analysis for Unisys Cobol files variables that are not explicitly declared in CALL statements. This analysis adds implicit use of variables declared in the Common Storage Section to every CALL statement of the program being analyzed, as well as for its PROCEDURE DIVISION USING phrase. That could lead to superfluous data dependencies between the caller and called programs in case the called program does not use data from Common Storage.

Relaxed Parsing

The **Relaxed Parsing** option in the workspace verification options lets you verify a source file despite errors. Ordinarily, the parser stops at a statement when it encounters an error. Relaxed parsing tells the parser to continue to the next statement.

For code verified with relaxed parsing, Component Maker behaves as follows:

- Statements included in a component that contain errors are treated as CONTINUE statements and appear in component text as comments.
- Dummy declarations for undeclared identifiers appear in component text as comments.
- Declarations that are in error appear in component text as they were in the original program. Corrected declarations appear in component text as comments.
- Commented-out code is identified by an extra comment line: "Enterprise Analyzer assumption".

PERFORM Behavior for Micro Focus Cobol

For Micro Focus Cobol applications, use the PERFORM behavior option in the workspace verification options window to specify the type of PERFORM behavior the application was compiled for. You can select:

- **Stack** if the application was compiled with the PERFORM-type option set to allow recursive PERFORMS.
- **All exits active** if the application was compiled with the PERFORM-type option set to not allow recursive PERFORMS.

For non-recursive PERFORM behavior, a COBOL program can contain PERFORM mines. In informal terms, a PERFORM mine is a place in a program that can contain an exit point of some active but not current PERFORM during program execution.

The program below, for example, contains a mine at the end of paragraph C. When the end of paragraph C is reached during PERFORM C THRU D execution, the mine "snaps" into action: control is transferred to the STOP RUN statement of paragraph A.

```
A.
  PERFORM B THRU C.
  STOP RUN.
 B.
  PERFORM C THRU D.
 C.
  DISPLAY 'C'.
 * mine
 D.
  DISPLAY 'D'.
```

Setting the compiler option to allow non-recursive PERFORM behavior where appropriate allows the Enterprise Analyzer parser to detect possible mines and determine their properties. That, in turn, lets Component Maker analyze control flow and eliminate dead code with greater precision. To return to our example, the mine placed at the end of paragraph C snaps each time it is reached: such a mine is called stable. Control never falls through a stable mine. Here it means that the code in paragraph D is unreachable.

Keep Legacy Copybooks Extraction Option

Select **Keep Legacy Copybooks** in the General extraction options for Cobol if you want Component Maker not to generate modified copybooks for the component. Component Maker issues a warning if including the original copybooks in the component would result in an error.

Example 1:

```
[COBOL]
01 A PIC X.
PROCEDURE DIVISION.
COPY CP.
[END-COBOL]
[COPYBOOK CP.CPY]
STOP RUN.
DISPLAY A.
[END-COPYBOOK CP.CPY]
```

For this example, Component Maker issues a warning for an undeclared identifier after Dead Code Elimination.

Example 2:

```
[COBOL]
PROCEDURE DIVISION.
COPY CP.
STOP RUN.
P.
[END-COBOL]
[COPYBOOK CP.CPY]
DISPLAY "QA is out there"
STOP RUN.
PERFORM P.
[END-COPYBOOK CP.CPY]
```

For this example, Component Maker issues a warning for an undeclared paragraph after Dead Code Elimination.

Example 3:

```
[COBOL]
working-storage section.
```

```
copy file.
PROCEDURE DIVISION.
p1.
  move 1 to a.
p2.
  display b.
  display a.
p3.
  stop run.
[END-COBOL]
[COPYBOOK file.cpy]
01 a pic 9.
01 b pic 9.
[END-COPYBOOK file.cpy]
```

For this example, the range component on paragraph p2 looks like this:

```
[COBOL]
WORKING-STORAGE SECTION.
  COPY FILE1.
  LINKAGE SECTION.
  PROCEDURE DIVISION USING A.
[END-COBOL]
while, with the option turned off, it looks like this:
[COBOL]
WORKING-STORAGE SECTION.
  COPY FILE1-A$RULE-0.
  LINKAGE SECTION.
  COPY FILE1-A$RULE-1.
[END-COBOL]
```

That is, turning the option on overrides the splitting of the copybook file into two files. Component Maker issues a warning if that could result in an error.

How Parameterized Slices Are Generated for Cobol Programs

The specifications for input and output parameters are:

- Input

  A variable of an arbitrary level from the LINKAGE section or PROCEDURE DIVISION USING is classified as an input parameter if one or more of its bits are used for reading before writing.

  A system variable (field of DFHEIB/DFHEIBLK structures) is classified as an input parameter if the **Create CICS Program** option is turned off and the variable is used for writing before reading.
- Output

  A variable of an arbitrary level from the LINKAGE section or PROCEDURE DIVISION USING is classified as an output parameter if it is modified during component execution.

  A system variable (a field of DFHEIB/DFHEIBLK structures) is classified as an output parameter if the **Create CICS Program** option is turned off and the variable is modified during component execution.
- For each input parameter, the algorithm finds its first usage (it does not have to be unique, the algorithm processes all of them), and if the variable (parameter from the LINKAGE section) is used for reading, code to copy its value from the corresponding field of BRE-INPUT-STRUCTURE is inserted as close to this usage as possible.
- The algorithm takes into account all partial or conditional assignments for this variable before its first usage and places PERFORM statements before these assignments.

  If a PERFORM statement can be executed more than once (as in the case of a loop), then a flag variable (named BRE-INIT-COPY-FLAG-[<*n*>] of the type PIC 9 VALUE 0 is created in the WORKING-

STORAGE section, and the parameter is copied into the corresponding variable only the first time this PERFORM statement is executed.

- For all component exit points, the algorithm inserts code to copy all output parameters from working-storage variables to the corresponding fields of BRE-OUTPUT-STRUCTURE.

  Variables of any level (rather than only 01-level structures together with all their fields) can act as parameters. This allows exclusion of unnecessary parameters, making the resulting programs more compact and clear.

  For each operator for which a parameter list is generated, the following transformations are applied to the entire list:

  - All FD entries are replaced with their data descriptions.
  - All array fields are replaced with the corresponding array declarations.
  - All upper-level RENAMES clauses are replaced with the renamed declarations.
  - All upper-level REDEFINES clauses with an object (including the object itself, if it is present in the parameter list) are replaced with a clause of a greater size.
  - All REDEFINES and RENAMES entries of any level are removed from the list.
  - All variable-length arrays are converted into fixed-length of the corresponding maximal size.
  - All keys and indices are removed from array declarations.
  - All VALUE clauses are removed from all declarations.
  - All conditional names are replaced with the corresponding data items.

Setting a Specialization Variable to Multiple Values

For Domain-Based Componentization, Component Maker lets you set the specialization variable to a range of values (between 1 and 10 inclusive, for example) or to multiple values (not only CHECK but CREDIT-CARD, for example). You can also set the variable to all values not in the range or set of possible values (every value but CHECK and CREDIT-CARD, for example).

Component Maker uses multiple values to predict conditional branches intelligently. In the following code fragment, for example, the second IF statement cannot be resolved with a single value, because of the two conflicting values of Z coming down from the different code paths of the first IF. With multiple values, however, Component Maker correctly resolves the second IF, because all the possible values of the variable at the point of the IF are known:

```
IF X EQUAL Y
  MOVE 1 TO Z
ELSE
  MOVE 2 TO Z
DISPLAY Z.
IF Z EQUAL 3
  DISPLAY "Z=3"
ELSE
  DISPLAY "Z<>3"
```

Keep in mind that only the following COBOL statements are interpreted with multiple values:

- COMPUTE
- MOVE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

That is, if the input of such a statement is defined, then, after interpretation, its output can be defined as well.

Single-Value Example:

```
MOVE 1 TO Y.
MOVE 1 TO X.
ADD X TO Y.
DISPLAY Y.
IF Y EQUAL 2 THEN...
```

In this fragment of code, the value of Y in the IF statement (as well as in DISPLAY) is known, and so the THEN branch can be predicted.

Multiple-Value Example:

```
IF X EQUAL 0
  MOVE 1 TO Y
ELSE
  MOVE 2 TO Y.
ADD 1 TO Y.
IF Y = 10 THEN... ELSE...
```

In this case, Component Maker determines that Y in the second IF statement can equal only 2 or 3, so the statement can be resolved to the ELSE branch.

The statement interpretation capability is available only when you define the specialization variable "positively" (as equalling a range or set of values), not when you define the variable "negatively" (as not equalling a range or set of values).

Arithmetic Exception Handling

For Cobol, the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements can have ON SIZE ERROR and NOT ON SIZE ERROR phrases. The phrase ON SIZE ERROR contains an arithmetic exception handler.

Statements in the ON SIZE ERROR phrase are executed when one of the following arithmetic exception conditions take place:

- The value of an arithmetic operation result is larger than the resultant-identifier picture size.
- Division by zero.
- Violation of the rules for the evaluation of exponentiation.

For MULTIPLY arithmetic statements, if any of the individual operations produces a size error condition, the statements in the ON SIZE ERROR phrase is not executed until all of the individual operations are completed.

Control is transferred to the statements defined in the phrase NOT ON SIZE ERROR when a NOT ON SIZE ERROR phrase is specified and no exceptions occurred. In that case, the ON SIZE ERROR is ignored.

Component Maker specialization processes an arithmetic statement with exception handlers in the following way:

- If a (NOT) ON SIZE ERROR condition occurred in some interpreting pass, then the arithmetic statement is replaced by the statements in the corresponding phrase.
- Those statements will be interpreted at the next pass.