



# Creating Components





IBM Asset Transformation Workbench  
v1.1



# Creating Components

Note:

Before using this information and the product it supports, read the information in “Notices.”

**First Edition (February 2005)**

This edition applies to IBM Asset Transformation Workbench (product number 5724-L54) and to all subsequent releases and modifications until otherwise indicated in new editions.

For the latest information about this product, please refer to the Release Notes.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of IBM. Information in this document is subject to change without notice and is not guaranteed to be error-free.

You can order publications through your IBM representative or the IBM branch office serving your locality. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Licensed Materials - Property of IBM.

Product Reference: IBM Asset Transformation Workbench v1.1

Document Reference: REL7.3.07.DOC06.A

**© 2005 Copyright International Business Machines Corporation. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

**© 2004, 2005 Relativity Technologies, Inc. All rights reserved.**

RescueWare is a registered trademark of Relativity Technologies, Inc. All other brands mentioned in this document are trademarks or registered trademarks of their respective holders.

# Contents

## Preface

<i>Audience</i> . . . . .	ix
<i>Organization</i> . . . . .	x
<i>Conventions</i> . . . . .	xi
<i>Related Manuals</i> . . . . .	xi
<i>Online Help</i> . . . . .	xii

## 1 Introducing Component Maker

<i>Componentization Methods</i> . . . . .	1-2
<i>Structure-Based Componentization (Cobol Only)</i> . . . . .	1-2
<i>Computation-Based Componentization (Cobol, Natural Only)</i> . . . . .	1-3
<i>Domain-Based Componentization (Cobol, PL/I Only)</i> . . . . .	1-3
<i>Event Injection (Cobol Only)</i> . . . . .	1-4
<i>Dead Code Elimination (DCE)</i> . . . . .	1-4
<i>Entry Point Isolation (Cobol Only)</i> . . . . .	1-4
<i>Componentization Outputs</i> . . . . .	1-5
<i>Starting Component Maker</i> . . . . .	1-6
<i>Component Maker Basics</i> . . . . .	1-7
<i>Getting Started in the Components Pane</i> . . . . .	1-7
<i>Working in the Components Pane</i> . . . . .	1-10
<i>Working with HyperView Lists</i> . . . . .	1-12

*Generating Audit Reports (Cobol Only)* . . . . . 1-13  
*Generating Coverage Reports (Cobol Only)* . . . . . 1-14  
*Exporting Logical Components* . . . . . 1-15  
*Generating CICS Components (Cobol Only)* . . . . . 1-15  
*What's Next?* . . . . . 1-15

**2 Setting Extraction Options**

*Opening the Extraction Options Windows* . . . . . 2-2  
*Setting Cobol Extraction Options* . . . . . 2-2  
    *General Options* . . . . . 2-2  
    *Interface Options* . . . . . 2-3  
    *Optimize Options* . . . . . 2-4  
    *Document Options* . . . . . 2-7  
    *Component Type Specific Options* . . . . . 2-8  
    *Component Conversion Options* . . . . . 2-12  
*Setting PL/I Extraction Options* . . . . . 2-13  
    *General Options* . . . . . 2-13  
    *Component Type-Specific Options* . . . . . 2-14  
    *Component Conversion Options* . . . . . 2-15  
*Setting Natural Extraction Options* . . . . . 2-15  
    *General Options* . . . . . 2-15  
    *Optimize Options* . . . . . 2-15  
    *Component Type-Specific Options* . . . . . 2-16  
    *Component Conversion Options* . . . . . 2-17  
*What's Next?* . . . . . 2-17

**3 Extracting Structure-Based Components**

*Understanding Ranges* . . . . . 3-1  
*Understanding Parameterized Slices* . . . . . 3-3  
*Extracting Structure-Based Cobol Components* . . . . . 3-6  
*Extracting Structure-Based PL/I Components (Future Release)* . . . . . 3-11  
*Extracting Structure-Based Natural Components (Future Release)* . . . . . 3-14  
*Extracting External Subroutines from Natural Programs (Future Release)* . . . . . 3-17  
*What's Next?* . . . . . 3-18

<b>4</b>	<b>Extracting Computation-Based Components</b>	
	<i>Understanding Variable-Based Extraction</i> . . . . .	4-2
	<i>Understanding Blocking</i> . . . . .	4-2
	<i>Extracting Computation-Based Cobol Components</i> . . . . .	4-3
	<i>Extracting Computation-Based Natural Components</i> . . . . .	4-7
	<i>What's Next?</i> . . . . .	4-10
<b>5</b>	<b>Extracting Domain-Based Components</b>	
	<i>Understanding Program Specialization in Simplified Mode</i> . . . . .	5-2
	<i>Understanding Program Specialization in Advanced Mode</i> . . . . .	5-5
	<i>Understanding Program Specialization "Lite"</i> . . . . .	5-6
	<i>Extracting Domain-Based Cobol Components</i> . . . . .	5-7
	<i>Extracting Domain-Based PL/I Components</i> . . . . .	5-14
	<i>What's Next?</i> . . . . .	5-17
<b>6</b>	<b>Injecting Events</b>	
	<i>Understanding Event Injection</i> . . . . .	6-2
	<i>Extracting Event-Injected Cobol Components</i> . . . . .	6-4
	<i>What's Next?</i> . . . . .	6-9
<b>7</b>	<b>Eliminating Dead Code</b>	
	<i>Generating Dead Code Statistics</i> . . . . .	7-2
	<i>Understanding Dead Code Elimination</i> . . . . .	7-2
	<i>Understanding Refactoring (Future Release)</i> . . . . .	7-3
	<i>Extracting Optimized Cobol Components</i> . . . . .	7-6
	<i>Extracting Optimized Natural Components</i> . . . . .	7-9
	<i>What's Next?</i> . . . . .	7-12
<b>8</b>	<b>Performing Entry Point Isolation</b>	
	<i>Extracting a Cobol Component with Entry Point Isolation</i> . . . . .	8-1
	<i>What's Next?</i> . . . . .	8-6

## **A Technical Details**

<i>Verification Options</i> . . . . .	A-1
<i>Use Special IMS Calling Conventions</i> . . . . .	A-1
<i>Override CICS Program Termination</i> . . . . .	A-2
<i>Support CICS HANDLE Statements</i> . . . . .	A-2
<i>Perform Unisys TIP and DPS Calls Analysis</i> . . . . .	A-2
<i>Perform Unisys Common-Storage Analysis</i> . . . . .	A-3
<i>Relaxed Parsing (Cobol and Natural Only)</i> . . . . .	A-4
<i>PERFORM Behavior for MicroFocus Cobol</i> . . . . .	A-5
<i>Keep Legacy Copybooks Extraction Option</i> . . . . .	A-6
<i>How Parameterized Slices Are Generated for Cobol Programs</i> . . . . .	A-8
<i>Setting a Specialization Variable to Multiple Values</i> . . . . .	A-10
<i>Arithmetic Exception Handling (COBOL Only)</i> . . . . .	A-12

## **Glossary**

## **Bibliography**

## **Notices**

## **Index**



## ***Preface***

**T**he IBM Asset Transformation Workbench (ATW) is a suite of PC-based software products for analyzing, re-architecting, and transforming legacy applications. The products are deployed in an integrated environment with access to a common repository of program objects. Repository models serve as the basis for a rich set of diagrams, reports, and other documentation.

The ATW suite consists of customizable modules that together address the needs of organizations at every stage of legacy application evolution — maintenance/enhancement, renovation, and modernization.

### ***Audience***

This guide assumes that you are a corporate Information Technology (IT) professional with a working knowledge of the legacy platforms you are using the product to analyze. If you are transforming a legacy application, you should also have a working knowledge of the target platform.

## Organization

This guide contains the following chapters:

- Chapter 1, “Introducing Component Maker,” provides an overview of component extraction methods and the Component Maker tool.
- Chapter 2, “Setting Extraction Options,” describes how you set Component Maker options for each of the component extraction methods and supported object types.
- Chapter 3, “Extracting Structure-Based Components,” describes how to extract a component based on a range of inline code.
- Chapter 4, “Extracting Computation-Based Components,” describes how to extract a component that contains all the code necessary to calculate the value of a variable at a particular point in a program.
- Chapter 5, “Extracting Domain-Based Components,” describes how to extract a component “specialized” on the values of one or more variables.
- Chapter 6, “Injecting Events,” describes how to adapt a legacy program to asynchronous, event-based programming models like MQ Series.
- Chapter 7, “Eliminating Dead Code,” describes how to extract a component from which unreferenced data items or unreachable procedural statements have been removed.
- Chapter 8, “Performing Entry Point Isolation,” describes how to extract a component based on one of multiple entry points in a legacy program.
- Appendix A, “Technical Details,” gives technical details of Component Maker behavior for a handful of narrowly focused verification and extraction options; for Cobol parameterized slice generation; for domain-based extraction when the specialization variable is set to multiple values; and for Cobol arithmetic exception handling.
- The Glossary defines the names, acronyms, and special terminology used in this guide.

## Conventions

This guide uses the following typographic conventions:

- **Bold type** — Indicates a specific area within the graphical user interface, such as a button on a screen, a window name, or a command or function.
- *Italic type* — Indicates a new term. Also indicates a document title. Occasionally, italic type is used for emphasis.
- Monospace type — Indicates computer programming code.
- **Bold monospace type** — Indicates input you type on the computer keyboard.
- **1A/1B, 2A/2B** — In task descriptions, indicates mutually exclusive steps: perform step A or step B, but not both.

## Related Manuals

This document is part of a complete set of ATW manuals. Together they provide all the information you need to get the most out of the system.

- *Getting Started* introduces ATW. This guide provides an overview of the workbench tools and discusses basic concepts. It describes how to install the product and how to manage licenses. It also describes how to use common product features.
- *Preparing Projects* describes how to set up ATW projects. This guide describes how to load applications in the repository and how to use reports and other tools to ensure that the entire application is available for analysis.
- *Analyzing Projects* describes how to analyze applications at the project level. This guide describes how to create diagrams of applications and how to perform impact analysis across applications. It also describes how to estimate project complexity and effort, and how to create a project dictionary.
- *Analyzing Programs* describes how to analyze applications at the program level. This guide describes how to use HyperView tools to

view programs interactively and perform program analysis in stages. It also describes how to analyze procedure and data flows, search the repository, and extract business rules with HyperView.

- *Profiling Projects* describes how to create and browse web-generated views of the repositories in your organization.
- *Parser Reference Manual* describes legacy constructions supported by Application Analyzer in reference format.
- *Architecture Reference Manual* describes legacy constructions supported by Application Architect in reference format.

## Online Help

In addition to the manuals provided with the system, you can learn about the product using the integrated online help. All GUI-based tools include a standard Windows **Help** menu.

You can display:

- The entire help system, with table of contents, index, and search tool, by selecting **Help: Help Topics**.
- Help about a particular ATW window by clicking the window and pressing the **F1** key.

Many ATW tools have *guides* that you can use to get started quickly in the tool. The guides are help-like systems with hyperlinks that you can use to access functions otherwise available only in menus and other program controls.

To open the guide for a tool, choose **Guide** from the **View** menu. Use the table of contents in the **Page** drop-down to navigate quickly to a topic.

# *Introducing Component Maker*



**A** *component* is a self-contained program that can be reused with other programs in modular fashion. The ATW Component Maker tool lets you “slice out” components from legacy applications — not only component executables but associated Cobol copybooks, PL/I includes, and Natural data areas as well.

You can export component files to your production environment, transform them into modern applications, or use them simply for documentation and analysis. All this without compromising the integrity of the original application.

Why use Component Maker? One reason is that legacy applications tend to be relatively unstructured. Modular programs are easier to understand, document, and maintain. Another is that structured applications are better suited for transformation into modern object-oriented programming languages.

## **Componentization Methods**

ATW offers a variety of methods for extracting components. Two of the methods described below — Dead Code Elimination and Entry Point Isolation — are optimization tools built into the main methods and offered separately in case you want to use them on a standalone basis. Each of the methods operates on program objects.

**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.

### **Structure-Based Componentization (Cobol Only)**

*Structure-Based Componentization* lets you build a component from a range of inline code — Cobol paragraphs, for example. Use traditional Structure-Based Componentization to generate a new component and its *complement*. A complement is a second component consisting of the original program minus the code extracted in the slice. Component Maker automatically places a call to the new component in the complement, passing it data items as necessary.

Alternatively, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. These standard object-oriented data interfaces make it easier to deploy the transformed component in modern service-oriented architectures.

#### **Specifying Multiple Ranges in a Cobol Extraction**

You typically repeat Structure-Based Componentization in incremental fashion until all of the modules you are interested in have been created. For Cobol programs, you can avoid doing this manually by specifying multiple ranges in the same extraction. Component Maker automatically processes each range in the appropriate order.

### **Computation-Based Componentization (Cobol and Natural Only)**

*Computation-Based Componentization* lets you build a component that contains all the code necessary to calculate the value of a variable at a particular point in a program — the value of a `DayOfTheWeek` variable, for example, where it is used to populate a report attribute or screen. As with structure-based componentization, you can generate parameterized slices that make it easy to deploy the transformed component in distributed architectures.

For Cobol programs, you can use a technique called *blocking* to produce smaller, better-defined parameterized components. Component Maker will not include in the slice any part of the calculation that appears before the blocked statement. Fields from blocked input statements are treated as input parameters of the component.

### **Domain-Based Componentization (Cobol and PL/I Only)**

*Domain-Based Componentization* lets you “specialize” a program based on the values of one or more variables. The specialized program is typically intended for reuse “in place” — in the original application but under new external circumstances.

After a change in your business practices, for example, a program that invokes processing for a “payment type” variable could be specialized on the value `PAYMENT-TYPE = "CHECK"`. Component Maker isolates every process dependent on the `CHECK` value to create a functionally complete program that processes check payments only.

Two modes of domain-based componentization are offered:

- The *simplified* mode treats the variable as a constant throughout the execution of the program — the variable’s value is “frozen in memory.” Operations that could change the value are ignored.
- The *advanced* mode sets the value of the variable only at a specified data port — subsequent operations can change the value, following the data and control flow of the program.

Use the simplified mode when you are interested only in the final value of a variable. Use the advanced mode when you need to account for data coming into a variable.

### **Event Injection (Cobol Only)**

*Event Injection* lets you adapt a legacy program to asynchronous, event-based programming models like MQ Series. You specify candidate locations for event calls (reads/writes, screen transactions, or subprogram calls, for example), the type of operation the event call performs (put or get), and the text of the message. For a put operation, for example, Component Maker builds a component that sends the message and any associated variable values to a queue, where the message can be retrieved by monitoring applications.

### **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:** You can perform Dead Code Elimination in batch mode, after batch registration and verification. For details, see *Preparing Projects* in the ATW documentation set.

#### **Refactoring Cobol Programs (Future Release)**

For Cobol programs, you can use an optimization method called *refactoring* to translate the program into a component with the same functionality and control flow but a simpler syntax structure. The component is said to be *defragmented*.

### **Entry Point Isolation (Cobol Only)**

*Entry Point Isolation* lets you build a component based on one of multiple entry points in a legacy program — an inner entry point in a Cobol program, for example. Component Maker extracts only the functionality and data definitions required for invocation from the selected point.

Entry Point Isolation is built into the main methods as an optional optimization tool. It's offered separately in case you want to use it on a standalone basis.



## Componentization Outputs

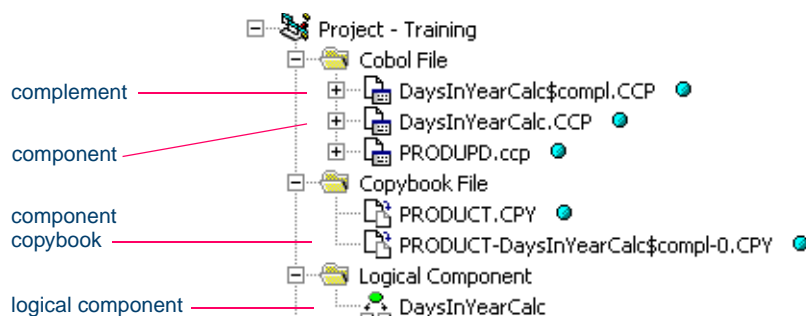
The first step in the componentization process, called *extraction*, creates the source files that comprise the component. It also creates an abstract repository object, or *logical component*, that gives you access to the source files in the workbench. The second step, called *conversion*, registers 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.
- On the other hand, if you are interested only in deploying the components in your production environment, you can skip the conversion step and avoid cluttering your repository. For information on how you export component source files to your file system, see [“Exporting Logical Components” on page 1-15](#).

Figure 1-1 shows how the componentization outputs are represented in the Repository Browser after conversion and verification of a structure-based Cobol component called DaysInYearCalc. PRODUPD is the program the component was extracted from.

Figure 1-1 *Componentization Objects After Conversion and Verification*



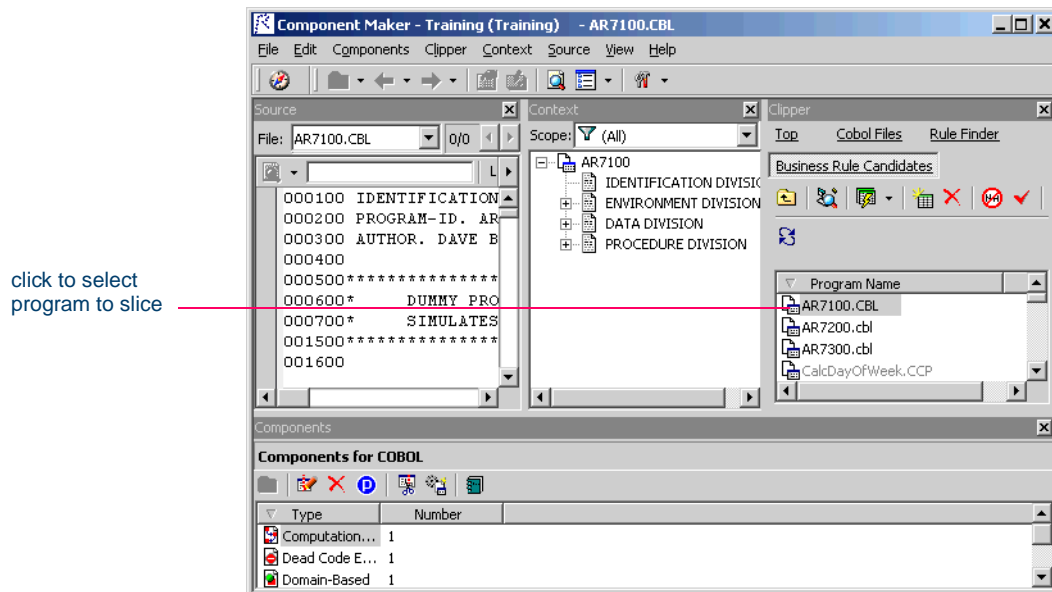
## Starting Component Maker

Component Maker is a HyperView-based tool that you can invoke on a standalone basis or from within HyperView itself:

- Start the tool in HyperView by selecting the program you want to slice in the ATW Repository Browser and choosing **Interactive Analysis** in the workbench **Analyze** menu. In the HyperView window, choose **Component Maker** in the **View** menu.
- Start the standalone tool by selecting the project that contains the programs you want to slice in the Repository Browser and choosing **Logical Components** in the workbench **Architect** menu. In the HyperView window, select the program you want to slice in the Clipper pane.

Figure 1-2 shows a typical configuration of the Component Maker window. For HyperView usage, see *Analyzing Programs* in the ATW document set.

Figure 1-2 Component Maker Window — Typical Configuration



## Component Maker Basics

The Component Maker window consists of a Source pane, Context pane, Clipper pane, Callie pane, Components pane, and Activity Log. You can hide a pane by clicking the close box in the upper righthand corner. Select the appropriate choice in the **View** menu to show the pane again.

### Getting Started in the Components Pane

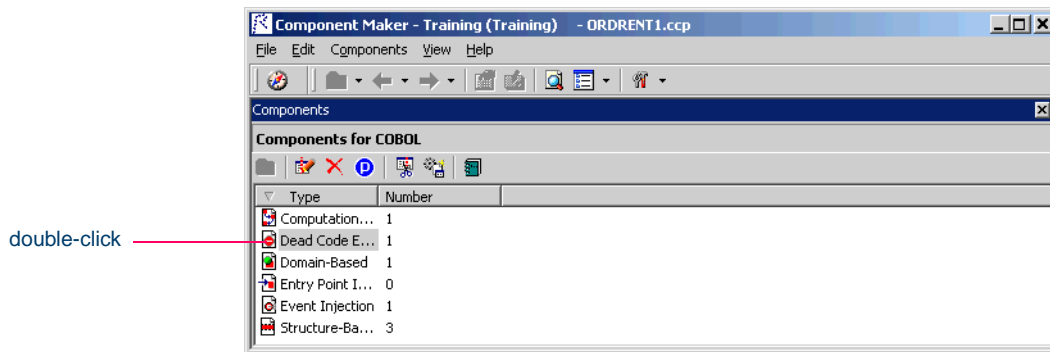
You do most of your work in Component Maker in the Components pane. To illustrate how you extract a logical component in the Components pane, let's look at the simplest task you can perform in Component Maker — Dead Code Elimination (DCE).

**Note:** The following exercise deliberately avoids describing the properties and options you can set for DCE. For detailed information on these features, see [Chapter 7, “Eliminating Dead Code.”](#)

#### To extract a DCE-based logical component:

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 1-3), double-click Dead Code Elimination.

Figure 1-3 *Components Pane*



- 2 The view shown in Figure 1-4 on page 1-8 opens. This view shows the DCE logical components created for the programs in the current


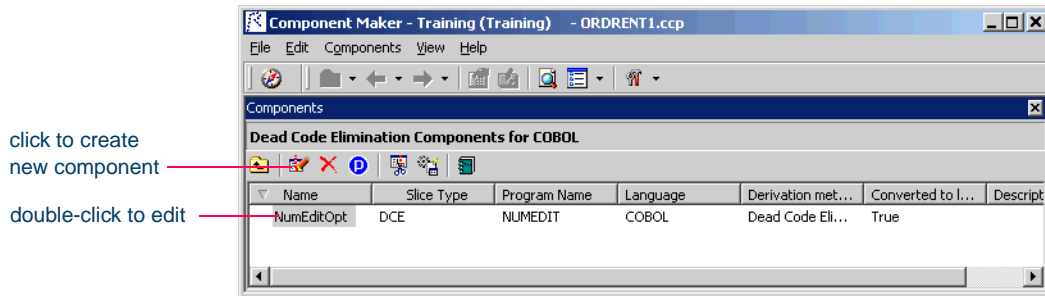
project. Click the  button on the tool bar to restrict the display to logical components created for the selected program.

Figure 1-4 *Components Pane — Dead Code Elimination View*





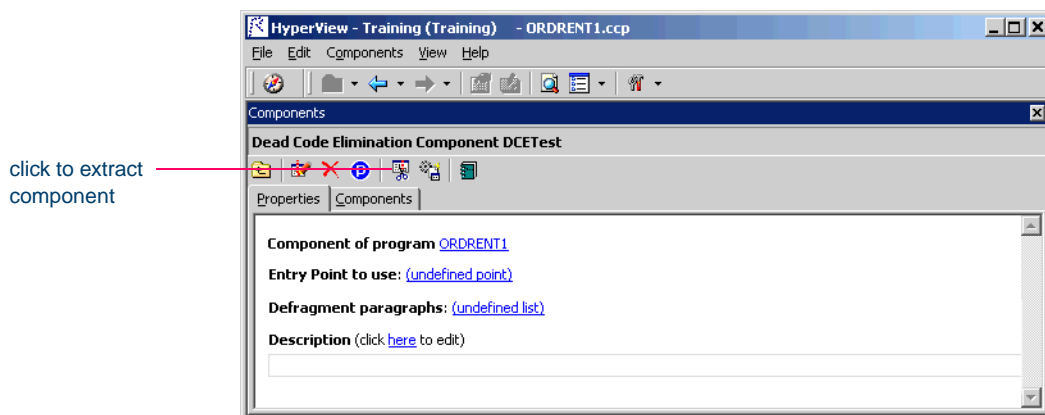
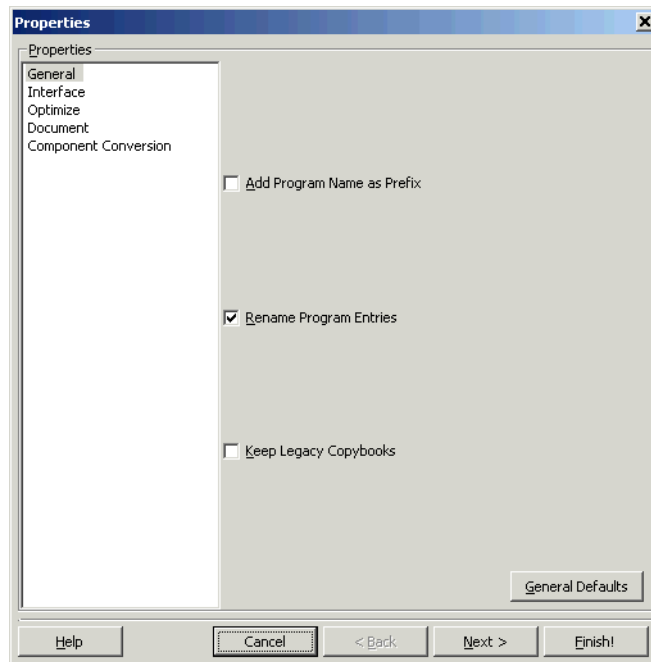
- 3A To create a new logical component, click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 1-5 opens. This view shows the properties you can set for the logical component. The defaults are acceptable. Click the  button to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.

Figure 1-5 *Components Pane — Properties Tab*



- 5 The dialog shown in Figure 1-6 opens. This dialog displays a series of panes that let you set extraction options for DCE. The defaults are acceptable. Click **Next** until you reach the last pane, then click **Finish**.

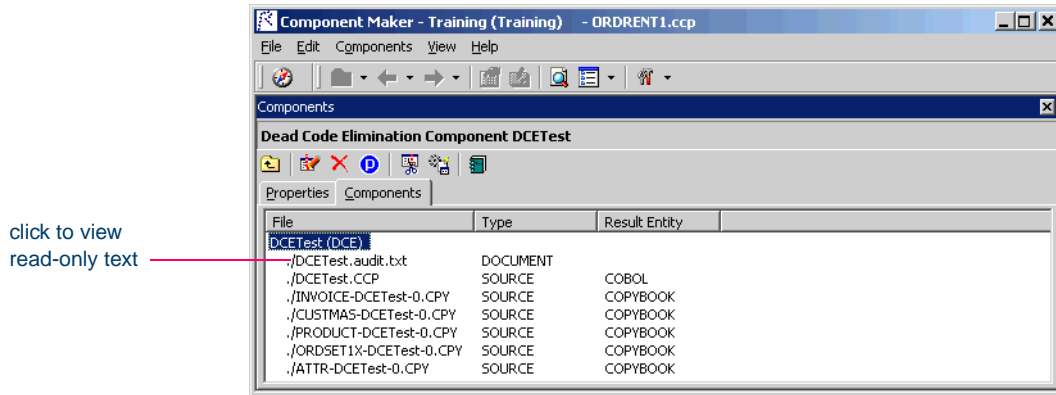
Figure 1-6 *Components Pane — Extraction Options Dialog*



**Note:** You may be warned that the extraction executed with errors or warnings. Click **Yes** to view the errors or warnings in the Activity Log.

- 6 Assuming the extraction executed without errors, the view shown in Figure 1-7 on page 1-10 opens. This view shows a list of the component source files that were generated for the logical component and an audit report (see [“Generating Audit Reports \(Cobol Only\)”](#) on [page 1-13](#)). Click an item in the list to view the read-only text for the item.


Figure 1-7 *Components Pane — Components Tab*



### **Working in the Components Pane**

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 — structure-based, computation-based, domain-based, and so on.
- 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.

Double-click an item in a view to access the next-level view in the hierarchy. Click the  button on the tool bar to navigate to the parent of the current view.



**Sorting Entries** Click a column heading in a view to sort the view entries by that column.

**Sizing Columns** Grab-and-drag the border of a column heading to increase or decrease the width of the column.

**Creating a Component** In the Clipper pane, select the program you want to slice. 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**.



**Setting Component Properties** In the Properties tab, click the **Component of program** property to navigate to the program in the Source pane. Usage of other properties varies by extraction method. For more information, see the chapter describing the method later in the manual.

**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**.


**Note:** Logical components are converted as well as extracted (see [“Componentization Outputs” on page 1-5](#)) if the **Convert Resulting Components to Legacy Objects** is set in the Component Conversion Options pane. For more information, see [“Component Conversion Options” on page 2-12](#).


**Viewing the Generated Files for a Component** In the Components tab, click an item in the list of generated files for the logical component to view the read-only text for the item.

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

**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**.

**Note:** Files are generated at extraction, not conversion.



**Restricting the Display to Program-Related Components** Click the  button on the tool bar to restrict the display to logical components of the selected program. The button is a toggle. Click it again to revert to the generic display.

**Deleting a Component** Select a logical component in the List view and click the  button on the tool bar to delete the component.

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

### Working with HyperView Lists

When you extract a logical component, Component Maker generates a list of constructs in the source program that have been included in the component. The list has the same name as the component.

To open the list, select the source program in the Clipper pane and click the  button next to the  button on the HyperView tool bar. Choose the list you want to open from the drop-down menu of available lists. The list opens in the HyperView List Browser (Figure 1-8). Click on a construct in the list to navigate to it in the source program.


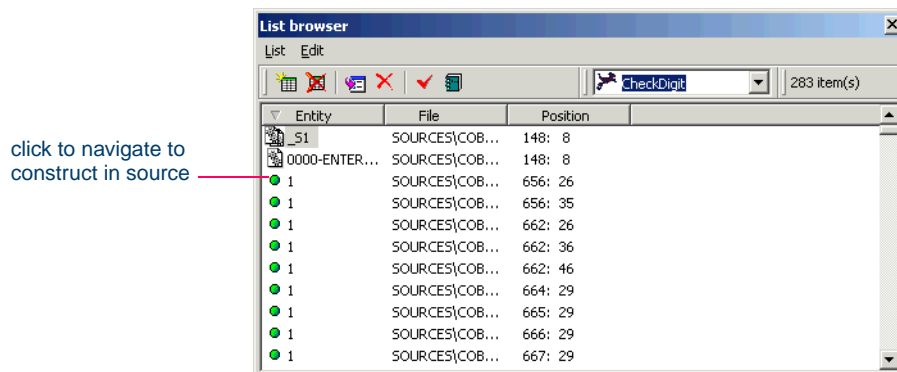
**Tip:** To place a check mark in the Source pane for each item in the list, click the  button on the List Browser tool bar.

Figure 1-8 *HyperView List Browser*





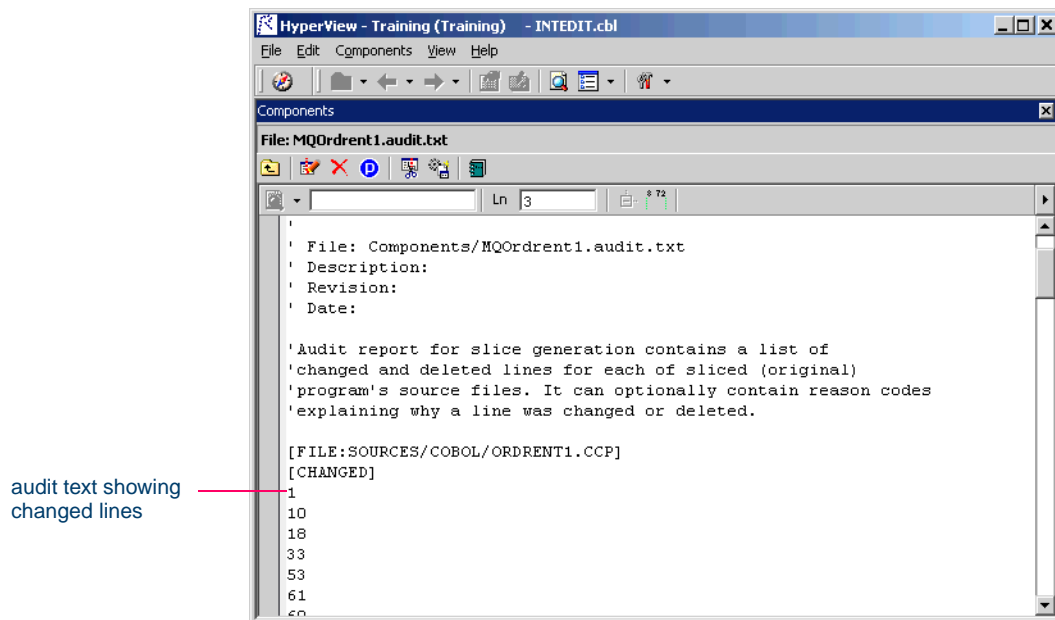
### Generating Audit Reports (Cobol Only)

An *audit report* (Figure 1-9) 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 read-only 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.

**Note:** For information on how you can set the audit report and reason code options, see [“Optimize Options” on page 2-4](#).


Figure 1-9 *Audit Report for MQOrdrent1*



## Generating Coverage Reports (Cobol Only)

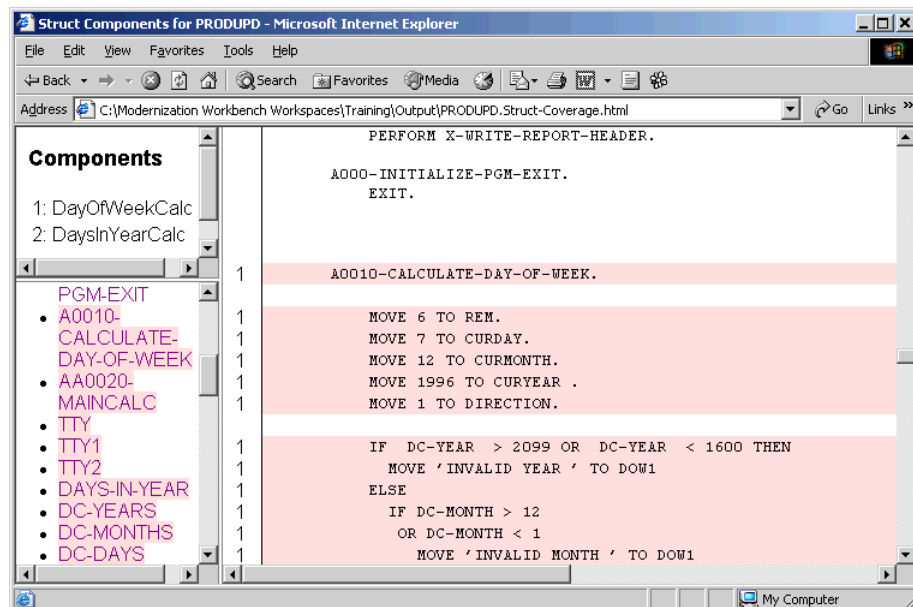
A coverage report (Figure 1-10) shows the extent to which a source program has been “componentized”:

- The top-left pane lists each component of a given type (structure-based, computation-based, and so on) 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 the  button 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-type-Coverage*. Double-click a report to view it in your Web browser.

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

Figure 1-10 Coverage Report for *PRODUPD* Structure-Based Components



## **Exporting Logical Components**

*Exporting* a logical component moves or copies the source files associated with it (including any complement or copybooks) from the ATW area to a specified location on your file system.

In the Repository Browser Logical Component folder, click the logical component whose source files you want to export, then choose **Export** in the **Architect** menu. A standard dialog appears where you can specify the destination for the source files. Place a check mark next to **Move files** if you want to move the files rather than copy them.

## **Generating CICS Components (Cobol Only)**

Component Maker let you generate structure- and computation-based Cobol components as CICS programs, with COMMAREAS for parameter exchange. That means the component can be called through a CICS LINK or by some other middleware such as IBM's ECI.

A CICS component can be run directly on mainframes:

- The component's parameters — whether original (from USING) or created by Component Extraction — are packaged under the CICS variable DFHCOMMAREA. There is no PROCEDURE DIVISION USING phrase in the component.
- At all program points where the original program could exit, the component exits through a CICS RETURN statement — any STOP RUN is replaced by CICS RETURN.

To generate CICS components, choose **Create CICS Program** in [step 3 on page 2-3](#).

## **What's Next?**

That completes your introduction to Component Maker. Now let's look at how you set Component Maker options for each of the componentization methods and supported object types.

**1-16** Introducing Component Maker  
*What's Next?*

## Setting Extraction 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 (Cobol, PL/I, and Natural). Extraction options are project-based — they apply to every program in the current ATW project.

### ***Set Cobol Verification Options!***

For computation- and domain-based componentization of Cobol programs, and for structure-based componentization with parameterized slices, you *must* turn on the **Perform Program Analysis** option in the Verification tab of the Project Options window before verifying the program you want to slice. For more information, see Appendix A, “Technical Details,” and *Preparing Projects* in the ATW document set.

## **Opening the Extraction Options Windows**

You can set Component Maker extraction options in the standard Project Options window or in the extraction options dialog:

- To open the standard Project Options window, choose **Project Options** in the **View** menu. In the Project Options window, click the Component Maker tab.
- To open the extraction options dialog (Figure 2-1 on page 2-4), follow steps [1-5](#) on pages 1-7 through 1-9.

This manual describes the dialog version. Usage is identical for the standard Project Options window.

### **Restoring Option Defaults**

You can restore the default extraction option settings in either type of window by clicking the **Option Type Defaults** button, then choosing **Restore Defaults** in the drop-down menu. Choose **Save To** in the drop-down menu to save the option settings to a file. Choose **Load From** in the menu to restore the option settings from a file.

## **Setting Cobol Extraction Options**

This section describes generic and method-specific component extraction options for Cobol programs.

### **General Options**

General component extraction options determine how components are named, whether inner entry points are renamed, and whether Component Maker generates modified copybooks.

#### **To set General component extraction options:**

- 1 In the extraction options dialog, click General in the lefthand pane.
- 2 Select **Add Program Name as Prefix** if you want Component Maker to prepend the name of the sliced program to the component name

you specified when you created the component ([step 3A on page 1-8](#)), in the form *program\$component*.

- 3** Select **Rename Program Entries** if you want Component Maker to 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 ATW parser can verify the component successfully. Unset this option if you need to preserve the original names of the inner entry points.
- 4** Select **Keep Legacy Copybooks** if you want Component Maker not to 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. For technical examples, see [page A-6](#).

### **Interface Options**

Interface component extraction options determine whether structure- and computation-based components are generated as parameterized slices and/or CICS programs.

#### **To set Interface component extraction options:**

- 1** In the extraction options dialog for a structure- or computation-based extraction, click Interface in the lefthand pane.
- 2** Select **Generate Parameterized Components** if you want Component Maker to extract parameterized slices. For background, see [“Understanding Parameterized Slices” on page 3-3](#).

**Important:** If you select **Generate Parameterized Components** for a structure-based extraction, you *must* set the **Range Only** option in the Component Type Specific pane. See [step 6 on page 2-9](#).

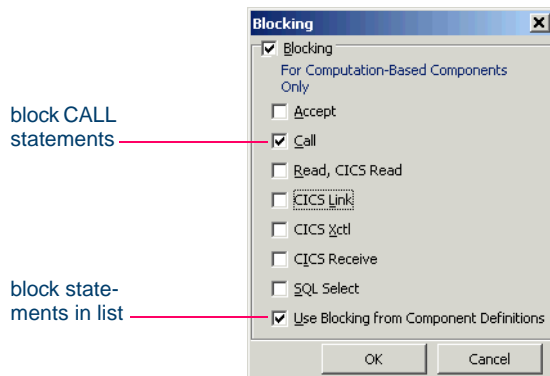
- 3** Select **Create CICS Program** if you want Component Maker to create COMMAREAS for parameter exchange in generated slices.

## 2-4 Setting Extraction Options *Setting Cobol Extraction Options*

For background, see [“Generating CICS Components \(Cobol Only\)” on page 1-15.](#)

- 4 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 (Figure 2-1). For background, see [“Understanding Variable-Based Extraction” on page 4-2.](#)

Figure 2-1 *Blocking Dialog*



### **Optimize Options**

Optimize component extraction options determine how Component Maker performs Dead Code Elimination and Cobol refactoring. For more information, see [Chapter 7, “Eliminating Dead Code.”](#)

#### **To set Optimize component extraction options:**

- 1 In the extraction options dialog, click Optimize in the lefthand pane.
- 2 Select **Optimize Code** to enable the code optimization options:
  - Select **Remove Unreachable Code** if you want Component Maker to remove unreachable procedural statements.
  - Select **Preserve Original Paragraphs** if you want Component Maker to generate paragraph labels even for paragraphs that are



not actually used in the source code — in particular, 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. For more information, see [“Understanding Refactoring \(Future Release\)” on page 7-3](#).

**3** In the Refactoring pane:

- Select **Roll-Up Nested IFs** if you want Component Maker to 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)
```

- Select **Remove Redundant NEXT SENTENCE** if you want Component Maker to 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.
```

- Select **Replace Section PERFORMs by Paragraph PERFORMs** if you want Component Maker to replace PERFORM section statements by equivalent PERFORM paragraph statements.

- In the **Defragment** drop-down, select:
  - **None** if you do not want Component Maker to perform control flow defragmentation.
  - **Whole Program** if you want Component Maker to perform control flow defragmentation for the entire program.
  - **Automatically Chosen Paragraphs** if you want Component Maker to evaluate the degree of control flow fragmentation of every paragraph and choose which paragraphs to defragment.
  - **Paragraphs in Component Definitions** if you want Component Maker to perform control flow defragmentation for the paragraphs in the list specified in the Properties tab. For more information, see [step 6 on page 7-7](#).
- 4 In the Handle Unused Data Items pane, select:
  - **No changes** if you want Component Maker not to remove unused data items from the component.
  - **Remove Unused Level-1 Structures** if you want Component Maker to remove only unused level-1 structures, and then only if all their members are unused. If, in the following example, only B is used, only G is regarded as dead:

```
DEFINE DATA LOCAL
  1 #A
    2 #B
      3 #C
    2 #D
      3 #E
      3 #F
  1 #G
```

- **Remove Unused Any-Level Structures** if you want Component Maker to remove unused structures at any data level, if all their members are unused. For the example above, D, E, F, and G are regarded as dead.
- **Remove/Replace Unused Fields with FILLERS** if you want Component Maker to 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 structure would adversely affect memory distribution.

**Note:** If you select **Keep Legacy Copybooks** in [step 4 on page 2-3](#), Component Maker removes or replaces with FILLERs only unused *inline* data items.

## Document Options

Document component extraction options determine how Component Maker documents its work — whether it generates an audit report, includes option settings in the component header, and the like. Generated document files are listed in the Components tab, as described in [“Viewing the Generated Files for a Component” on page 1-11](#).

### **To set Document component extraction options:**

- 1 In the extraction options dialog, click Document in the lefthand pane.
- 2 Select **Generate Audit Report** if you want Component Maker to generate an audit report. Select **Include Reason Codes** if you want the report to contain reason codes explaining why a line was changed or deleted. For more information, see [“Generating Audit Reports \(Cobol Only\)” on page 1-13](#).

**Note:** Generating reason codes is very memory-intensive and may cause crashes for extractions from large programs.

- 3 Select **List Options in Component Header and in Separate Document** if you want Component Maker to 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*.
- 4 Select **Generate Support Comments** if you want Component Maker to 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.

- 5 Select **Emphasize Component/Include in Coverage Report** if you want Component Maker to generate a list of constructs extracted from the source program (see [“Working with HyperView Lists” on page 1-12](#)) and highlight extracted code in the Coverage Report (see [“Generating Coverage Reports \(Cobol Only\)” on page 1-14](#)).
- 6 In the Annotate Legacy Code pane, select:
  - **Comment-out Sliced-off Legacy Code** if you want Component Maker to retain but comment out unused code in the component source. In the **Comment Prefix** field, enter text (up to six characters) that you want to appear to the left of the commented-out lines.
  - **Mark Modified Legacy Code** if you want Component Maker to mark modified code in the component source. In the **Comment Prefix** field, enter text (up to six characters) that you want to appear to the left of the modified lines.

### ***Component Type Specific Options***

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

#### ***Structure Based Type-Specific Options***

Structure-based type-specific extraction options determine whether Component Maker performs “relaxed” extraction, generates a complement, and the like.

#### ***To set structure-based type-specific extraction options:***

- 1 In the extraction options dialog for a structure-based extraction, click Component Type Specific in the lefthand pane.
- 2 Select **Restrict User Ranges to PERFORMed Ones** if you want Component Maker not to extract paragraphs that do not have a corresponding PERFORM statement. This option is useful if you want to limit components created with the Paragraph Pair or Section methods to PERFORMed paragraphs. For background, see [“Understanding Ranges” on page 3-1](#).

- 3 Select **Ensure Consistent Access to External Resources** if you want Component Maker to monitor the integrity of data flow in the ranges you are extracting. If you select this option, for example, an extraction will fail if an SQL cursor used in the component is open in the complement.
- 4 Select **Dynamic Call** if you want Component Maker to generate in the complement a dynamic call to the component. The complement will call a string variable that must later be set outside the complement to the name of the component.
- 5 Select **Suppress Errors** if you want Component Maker to perform a “relaxed extraction,” in which errors that would ordinarily cause the extraction to fail are ignored, and comments describing the errors are added to the component source. This option is useful when you want to review extraction errors in component source.
- 6 Select **Range Only** if you want Component Maker not to generate a complement. You *must* set this option to generate parameterized slices. See [step 2 on page 2-3](#).

### **Computation-Based Type-Specific Options**

Computation-based type-specific extraction options determine whether Component Maker performs variable-based component extraction and generates an HTML extraction trace.

#### **To set computation-based type-specific extraction options:**

- 1 In the extraction options dialog for computation-based extraction, click Component Type Specific in the lefthand pane.
- 2 In the Variable/Statement Based pane, select **Variable** if you want Component Maker to perform variable-based component extraction. Select **Statement** if you want Component Maker to perform statement-based component extraction. For background, see [“Understanding Variable-Based Extraction” on page 4-2](#).

**Note:** Even if you select variable-based extraction, Component Maker performs statement-based extraction if the variable you slice on is not an input variable for its parent statement — if the statement writes to rather than reads from the variable.

- 3 Select **Generate HTML Trace** to generate an HTML file with an extraction trace. The trace has a name of the form *component.trace*. To view the trace, click the logical component for the extraction in the Repository Browser Logical Component folder. Double-click the trace file to view it in your Web browser.

### **Domain-Based Type-Specific Options**

Domain-based component extraction options determine whether Component Maker removes unused assignments, replaces variables with their values, and evaluates conditional logic in one or multiple passes.

#### **To set domain-based type-specific extraction options:**

- 1 In the extraction options dialog for a domain-based extraction, click Component Type Specific in the lefthand pane.
- 2 Select **VALUEs Initialize Data Items** if you want Component Maker to set variables declared with VALUE clauses to their initial values. Otherwise, VALUE clauses are ignored.
- 3 Select **Remove Unused Assignments** if you want Component Maker to exclude from the component assignments that cannot affect the computation — typically, an assignment after which the variable is not used until the next assignment or port.
- 4 Select **Replace Variables by Their Calculated Values** if you want Component Maker to substitute the calculated values of variables for the variables themselves.

**Tip:** Notice how the options in steps 3 and 4 can interact. If both options are set, then the first assignment in the following fragment will be removed:

```
MOVE 1 TO X.  
DISPLAY X.  
MOVE 2 TO X.
```

- 5 In the Iterative Processing pane, select **Single Pass** if you want Component Maker to evaluate conditional logic in one pass. Select **Multiple Pass** if you want Component Maker to evaluate conditional logic again after detecting dead branches. Because the ELSE branch

of the first IF below is dead, for example, the second IF statement can be resolved in a subsequent pass:

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

**Note:** Multi-pass processing is very resource-intensive, and not recommended for extractions from large programs.

- 6 In the **Maximum Number of Variable's Values** field, enter the maximum number of values to be calculated for each variable. Limit is 100. In the **Maximum Size of Variable to Be Calculated** field, enter the maximum size in bytes for each variable value to be calculated. The lower the maximums, the better performance and memory usage you can expect. For each setting, you are warned about variables for which the specified maximum is exceeded.

### ***Event Injection Type-Specific Options***

Event injection component extraction options determine the middleware template you want to use for event injection and the type of statement to execute in case of an error connecting to middleware.

#### ***To set event injection type-specific extraction options:***

- 1 In the extraction options dialog for event injection, click Component Type Specific in the lefthand pane.
- 2 In the **Use Middleware** drop-down, select:
  - MQ if you want Component Maker to use an IBM MQ Series template for event injection.
    - In the **Queue Manager** field, enter the name of the MQ Series queue manager.
    - In the **Target Queue Name** field, enter the name of the target queue.

## 2-12 Setting Extraction Options

### *Setting Cobol Extraction Options*

- In the Use MQPUT/MQPUT1 pane, select **MQPUT** to use the MQPUT method. Select **MQPUT1** to use the MQPUT1 method.
  - User Defined if you want Component Maker to use a user-defined template for event injection. In the **User Specified Event** field, enter the name of the event to inject at the specified injection points.
- 3 In the **Error Handling** drop-down, select the type of statement to execute in case of an error connecting to middleware.

### **Component Conversion Options**

Component conversion extraction options determine whether components are converted as well as extracted and, if so, whether existing repository objects for the component are preserved or replaced.

#### **To set Component Conversion extraction options:**

- 1 In the extraction options dialog, click Component Conversion in the lefthand pane.
- 2 Select **Convert Resulting Components to Legacy Objects** if you want Component Maker to convert as well as extract the logical component (see [“Componentization Outputs” on page 1-5](#)). In the Component Converter Conflicts pane, choose either:
  - **Keep Old Legacy Objects** if you want Component Maker to preserve existing repository objects for the component — copy-books, 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.
  - **Replace Old Legacy Objects** if you want Component Maker to replace existing repository objects for the 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 Object**.

## Setting PL/I Extraction Options

This section describes generic and method-specific component extraction options for PL/I programs.

### General Options

General component extraction options determine whether the layout of the sliced program is preserved, whether to expand include files, and how the component is named.

#### To set General component extraction options:

- 1 In the extraction options dialog, click General in the lefthand pane.
- 2 In the Generation Style pane, select:
  - **Plain Generation** if you want Component Maker to ignore the layout of the sliced program. Macros and include files are expanded in the component.
  - **Preserve Legacy Layout** if you want Component Maker to preserve the layout and commentary of the sliced program. Macros and include files are expanded in the component.
  - **Preserve Legacy Includes** if you want Component Maker not to generate modified program include files for the component. Macros are expanded, and the layout and commentary of the sliced program is preserved.
- 3 Select **Add Program Name as Prefix** if you want Component Maker to prepend the name of the sliced program to the component name you specified when you created the component ([step 3A on page 1-8](#)), in the form *program\$component*.

### **Component Type-Specific Options**

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

#### **Structure Based Type-Specific Options (Future Release)**

Structure-based type-specific extraction options determine whether Component Maker generates a parameterized slice.

##### **To set structure-based type-specific extraction options:**

- 1 In the extraction options dialog, click Component Type Specific in the lefthand pane.
- 2 Select **Generate Parameterized Components** if you want Component Maker to generate a parameterized slice. Enter the name of the input, output, and input/output structures to be generated in the appropriate fields. For background, see [“Understanding Parameterized Slices” on page 3-3](#).

#### **Dead Code Elimination Type-Specific Options**

Dead Code Elimination type-specific extraction options determine whether Component Maker removes unreachable top-level procedures and/or performs DCE against the entire project to which the selected program belongs.

##### **To set DCE type-specific extraction options:**

- 1 In the extraction options dialog, click Component Type Specific in the lefthand pane.
- 2 Select **Library Mode** if you want Component Maker to include in the component external procedures that are not reachable from the main procedures in the project.
- 3 Select **Componentize Whole Project** if you want Component Maker to perform Dead Code Elimination against the entire project to which the selected program belongs.

**Note:** This option is always selected if you choose not to use the library mode option described in [step 2 on page 2-14](#).

## **Component Conversion Options**

Component conversion extraction options determine whether components are converted as well as extracted and, if so, whether existing repository objects for the component are preserved or replaced. For details, see the description of the identical options for Cobol programs in [“Component Conversion Options” on page 2-12.](#)

## **Setting Natural Extraction Options**

This section describes generic and method-specific component extraction options for Natural programs.

### **General Options**

General component extraction options determine whether to use program include files rather than generate data structures inline.

#### **To set General component extraction options:**

- 1 In the extraction options dialog, click General in the lefthand pane.
- 2 Select **Preserve Legacy Includes** if you want Component Maker not to generate modified program include files for the component.

### **Optimize Options**

Optimize component extraction options determine how Component Maker performs Dead Code Elimination (DCE). For more information, see [Chapter 7, “Eliminating Dead Code.”](#)

#### **To set Optimize extraction options:**

- 1 In the extraction options dialog, click Optimize in the lefthand pane.
- 2 In the Handle Unused Data Items pane, select:
  - **No changes** if you want Component Maker not to remove unused data items.
  - **Remove Unused Level-1 Structures** if you want Component Maker to remove only unused level-1 structures, and then only

## 2-16 Setting Extraction Options

### Setting Natural Extraction Options

if all of their members are unused. If, in the following example, only B is used, only G is regarded as dead:

```
DEFINE DATA LOCAL
  1 #A
    2 #B
      3 #C
    2 #D
      3 #E
      3 #F
  1 #G
```

- **Remove Unused Any-Level Structures** if you want Component Maker to remove unused structures at any data level, if all of their members are unused. For the example above, D, E, F, and G are regarded as dead.

**Notes:** **Remove Unused Any-Level Structures** is not available if you choose to comment out rather than remove dead code in [step 2 on page 2-17](#).

If you select **Preserve Legacy Includes** in [step 2 on page 2-15](#), Component Maker removes only unused *inline* data items.

### Component Type-Specific Options

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

#### Structure-Based Type-Specific Options (Future Release)

Structure-based type-specific extraction options determine whether to use program data areas rather than generate data structures inline, and whether to generate MAP files for a component.

#### To set structure-based type-specific extraction options:

- 1 In the extraction options dialog, click Component Type Specific in the lefthand pane.

- 2 Select **Use DA** if you want Component Maker to use program data areas rather than generate data structures inline.
- 3 Select **Generate Map File** if you want Component Maker to general a MAP file for the component.

### ***Dead Code Elimination Type-Specific Options***

Dead Code Elimination type-specific extraction options determine whether Component Maker comments out or removes dead code.

#### ***To set DCE type-specific extraction options:***

- 1 In the extraction options dialog, click Component Type Specific in the lefthand pane.
- 2 Select **Comment** if you want Component Maker to comment out dead code in the component. Select **Remove** if you want Component Maker to remove dead code from the component.

### ***Component Conversion Options***

Component conversion extraction options determine whether components are converted as well as extracted and, if so, whether existing repository objects for the component are preserved or replaced. For details, see the description of the identical options for Cobol programs in [“Component Conversion Options” on page 2-12.](#)

### ***What's Next?***

That completes our survey of Component Maker options. Now let's look at how you use Component Maker to perform each of the componentization methods.

**2-18** Setting Extraction Options

*What's Next?*

## ***Extracting Structure-Based Components***



**S**tructure-Based Componentization lets you build a component from a range of inline code — Cobol paragraphs, for example. Use traditional structure-based componentization to generate a new component and its *complement*. A complement is a second component consisting of the original program minus the code extracted in the slice. Component Maker automatically places a call to the new component in the complement, passing it data items as necessary.

Alternatively, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. These standard object-oriented data interfaces make it easier to deploy the transformed component in modern service-oriented architectures.

### ***Understanding Ranges***

When you extract a structure-based component from a program, you specify the *range* of code you want to include in the component — for

Cobol programs, a range of paragraphs; for PL/I programs, a procedure; for Natural programs, a range of subroutines.

#### ***Specifying Multiple Ranges in a Cobol Extraction***

You typically repeat Structure-Based Componentization in incremental fashion until all of the modules you are interested in have been created. For Cobol programs, you can avoid doing this manually by specifying multiple ranges in the same extraction. Component Maker automatically processes each range in the appropriate order. No complements are generated.

#### ***Specifying Ranges for Cobol Programs***

For Cobol programs, you specify the paragraphs in the range in one of three ways:

- Select a *Paragraph Perform* statement to set the range to the performed paragraph or paragraphs. For COBOL, this set includes each paragraph in the execution path between the first and last paragraphs in the range, except when control is transferred by a PERFORM statement or by an implicit RETURN-from-PERFORM statement.
- Select a *Pair of Paragraphs* to set the range to the selected paragraphs. You are responsible for ensuring a continuous flow of control from the first to the last paragraph in the range.
- Select a *Section* to set the range to the paragraphs in the section.

#### ***About the GOTO Statement in a Cobol Complement***

For traditional structure-based COBOL components, Component Maker inserts in the complement the labels of the first and last paragraphs in the range. The first paragraph is replaced in the complement with a CALL statement followed by a GOTO statement. The last paragraph is always empty.

The GOTO statement transfers control to the last paragraph. If the GOTO statement and its target paragraph are not required to ensure correct call flow, they are omitted.



### ***Specifying Ranges for PL/I Programs (Future Release)***

For PL/I programs, you specify an internal procedure that Component Maker extracts as an external procedure. The slice contains the required parameters for global variables.

### ***Specifying Ranges for Natural Programs (Future Release)***

For Natural programs, you specify a range of internal subroutines. You are responsible for ensuring a continuous flow of control from the first to the last subroutine in the range.

Use a separate method called *External Subroutine Componentization* to extract a single internal subroutine as an external subroutine. For more information, see [“Extracting External Subroutines from Natural Programs \(Future Release\)”](#) on page 3-17.

## ***Understanding Parameterized Slices***

For Cobol and PL/I programs, you can generate *parameterized slices*, in which the input and output variables required by the component are organized in group-level structures. The component contains all the code required for input/output operations.

### ***How to Extract a Parameterized Slice***

Select the **Generate Parameterized Components** option in the extraction options dialog to extract a parameterized slice (for Cobol, see [step 2 on page 2-3](#); for PL/I, see [step 2 on page 2-14](#)). Note that you cannot generate a complement for a parameterized Cobol slice

### ***Cobol Naming Conventions***

- Component input structures have names of the form BRE-INP-*STRUCT-NAME*. Input fields have names of the form BRE-I-*FIELD-NAME*.
- Component Output structures have names of the form BRE-OUT-*STRUCT-NAME*. Output fields have names of the form BRE-O-*FIELD-NAME*.

**3-4** Extracting Structure-Based Components  
*Understanding Parameterized Slices*

***PL/I Naming Conventions (Future Release)***

For PL/I, component structures can have any name you want. Field names are retained. An input/output structure may also be generated.

***Example***

Consider a COBOL program that contains the following structures:

```
WORKING-STORAGE SECTION.  
    01 A  
        03 A1  
        03 A2  
    01 B  
        03 B1  
        03 B2  
        03 B4
```

Suppose that only A1 has been determined by Component Maker to be an input parameter, and only B1 and B2 to be output parameters. Suppose further that the component is extracted with input and output data structures that use the default names, BRE-INP-INPUT-STRUCTURE and BRE-OUT-OUTPUT-STRUCTURE, respectively, and the default Optimization options.

The component contains the following code:

```
WORKING-STORAGE SECTION.  
    01 A  
        03 A1  
        03 A2  
    01 B  
        03 B1  
        03 B2  
        03 B4  
LINKAGE SECTION.  
    01 BRE-INP-INPUT-STRUCTURE  
        03 BRE-I-A  
            06 BRE-I-A1
```

```
01 BRE-OUT-OUTPUT-STRUCTURE
    03 BRE-O-B
        06 BRE-O-B1
        06 BRE-O-B2
PROCEDURE DIVISION
    USING BRE-INP-INPUT-STRUCTURE BRE-OUT-OUTPUT-
STRUCTURE.
BRE-INIT-SECTION SECTION.
    PERFORM BRE-COPY-INPUT-DATA.
    .....
    ....(Business Logic)....
    .....
    *ATW added statement
    GO TO BRE-EXIT-PROGRAM.
BRE-EXIT-PROGRAM-SECTION SECTION.
    BRE-EXIT-PROGRAM.
        PERFORM BRE-COPY-OUTPUT-DATA.
        GOBACK.
BRE-COPY-INPUT-DATA.
    MOVE BRE-I-A TO A.
BRE-COPY-OUTPUT-DATA.
    MOVE B TO BRE-O-B.
```

***Set Cobol Verification Options!***

For parameterized structure- and computation-based componentization of Cobol programs, you *must* turn on not only the **Perform Program Analysis** option in the Verification tab of the Project Options window, but also the **Enable Parameterization of Components** option in the same dialog. For information on how to set verification options, see *Preparing Projects* in the ATW document set.

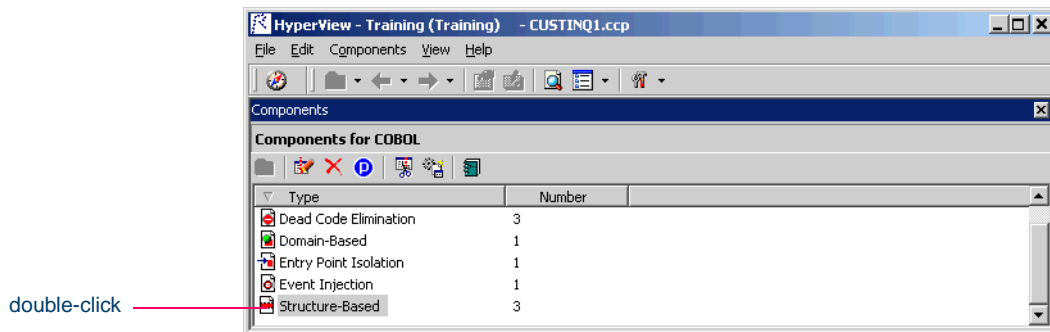
## Extracting Structure-Based Cobol Components

This section describes how to perform Structure-Based Componentization for Cobol programs.

### To extract a structure-based Cobol component:

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 3-1), double-click Structure-Based.

Figure 3-1 Components Pane



- 2 The view shown in Figure 3-2 on page 3-7 opens. This view shows the structure-based logical components created for the programs in the current project.


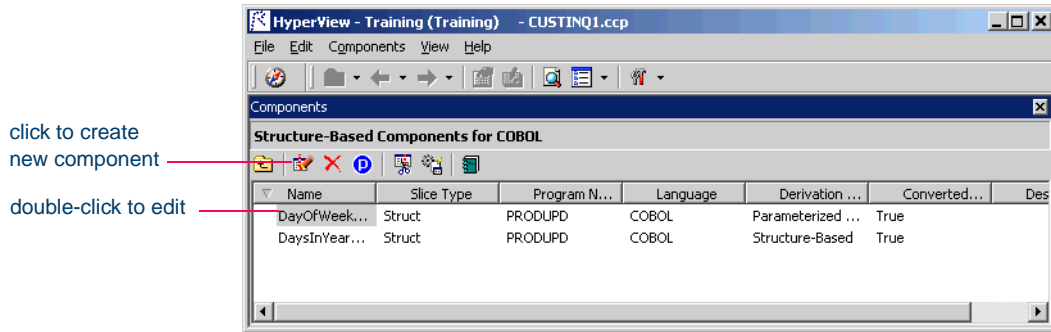
**Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.

Figure 3-2 *Components Pane — Structure-Based View*




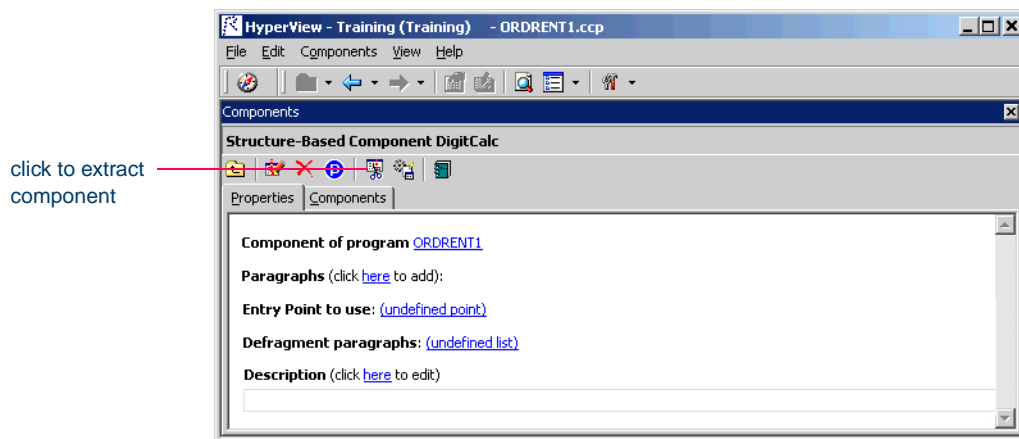
- 3A** To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B** To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4** The view shown in Figure 3-3 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 3-3 *Components Pane — Properties Tab (Cobol)*



- 5 In the **Paragraphs** field, click the **here** link. Choose one of the following methods in the pop-up menu. For background on the methods, see [“Understanding Ranges” on page 3-1](#).
  - **Paragraph Perform** to set the range to the paragraph or paragraphs performed by the selected PERFORM statement. Select the PERFORM statement in the Source pane, then click the link for the current selection and choose **Set** in the pop-up menu.
  - **Pair of Paragraphs** to set the range to the selected paragraphs. Select the first paragraph in the pair in the Source pane, then click the link for the current selection in the **From** field and choose **Set** in the drop-down menu. Select the second paragraph in the pair, then click the link for the current selection in the **To** field and choose **Set** in the pop-up menu.

**Tip:** You can set the **From** and **To** fields to the same paragraph.

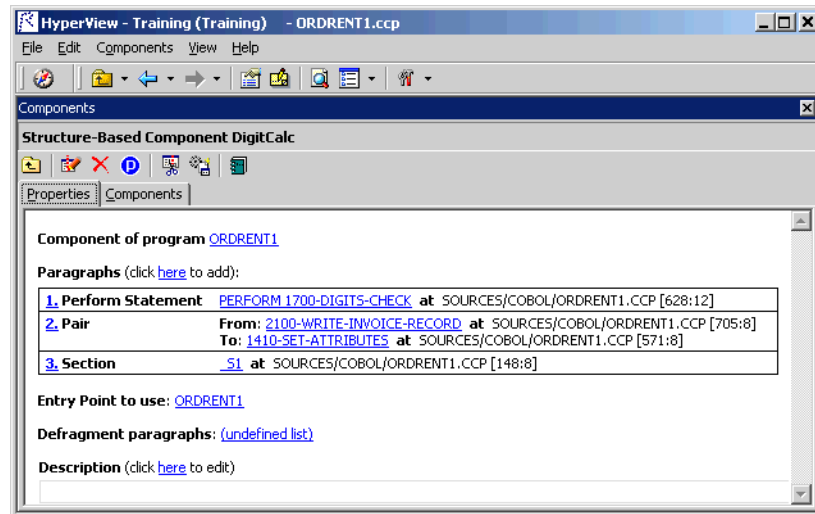
- **Section** to set the range to the paragraphs in the section. Select the section in the Source pane, then click the link for the current selection and choose **Set** in the pop-up menu.

**Note:** To delete a range, select the link for the numeral that identifies the range and choose **Delete** in the pop-up menu. To unset a PERFORM, paragraph, or section, click it and choose **Unset** in the pop-up menu. To navigate quickly to a PERFORM, paragraph, or section in the source, click it and choose **Locate** in the pop-up menu.

- 6 Repeat [step 5](#) for each range you want to extract. You can use any combination of methods. For background, see [“Specifying Multiple Ranges in a Cobol Extraction” on page 3-2](#). Figure 3-4 on page 3-9 shows how the properties tab might look for a multi-range extraction.

**Note:** The component for each range in a multi-range extraction has a name of the form *component*\$*n*, where *n* represents the order in which the component was generated.

Figure 3-4 Properties Tab — Multi-Range Cobol Extraction



- 7 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. For background, see [Chapter 8, “Performing Entry Point Isolation.”](#) To unset an entry point, click it and choose **Unset** in the pop-up menu.
  - 8 In the **Defragment paragraphs** field, click the link for the current selection and choose the list of paragraphs to defragment in the pop-up menu. For background, see [“Understanding Refactoring \(Future Release\)” on page 7-3.](#)
- Note:** To create a new list of paragraphs to defragment, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.
- 9 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

**3-10** Extracting Structure-Based Components  
*Extracting Structure-Based Cobol Components*

and in the Description property for the logical component repository object.


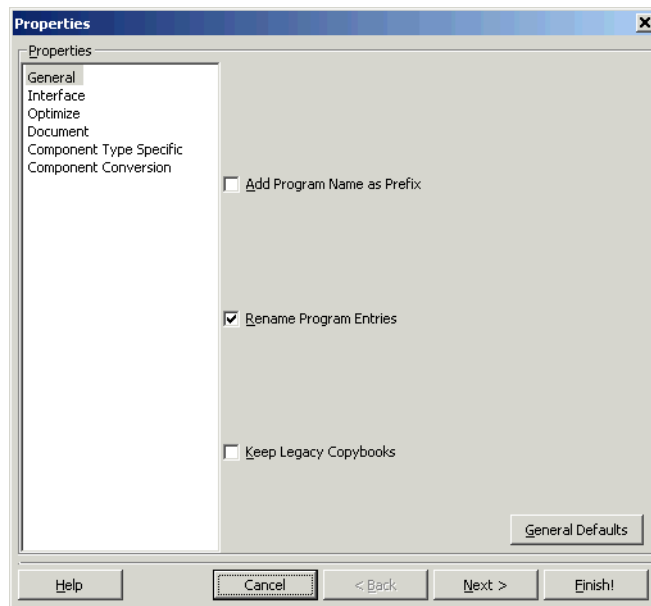
- 10 Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
- 11 The dialog shown in Figure 3-5 opens. This dialog displays a series of panes that let you set extraction options for Structure-Based Componentization. For usage information, see [“Setting Cobol Extraction Options” on page 2-2](#). When you are satisfied with your choices, click **Finish**.

Figure 3-5 *Components Pane — Extraction Options Dialog (Cobol)*

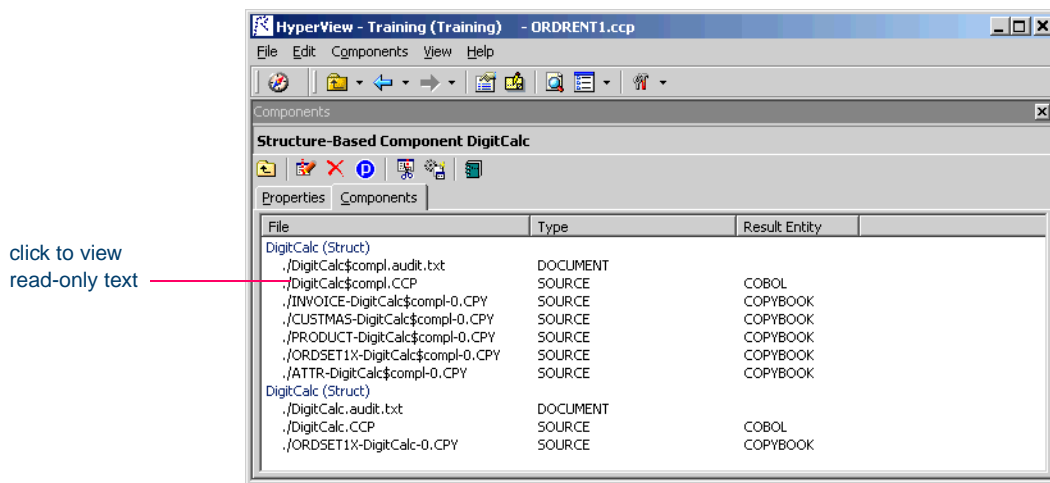


- 12 Component Maker performs the extraction. You are notified that the extraction is complete. If the extraction completed without errors or warnings, click **OK** to continue. If the extraction completed with errors or warnings, click **Yes** to view the errors or warnings in the Activity Log. Otherwise, click **No**.



- 13 Assuming the extraction executed without errors, the view shown in Figure 3-6 opens. This view shows a list of the component source files that were generated for the logical component and an audit report if you requested one (see [“Generating Audit Reports \(Cobol Only\)”](#) on page 1-13). Click an item in the list to view the read-only text for the item.

Figure 3-6 *Components Pane — Components Tab*




## Extracting Structure-Based PL/I Components (Future Release)

This section describes how to perform Structure-Based Componentization for PL/I programs.

### To extract a structure-based PL/I component:

- 1 Start Component Maker as described in [“Starting Component Maker”](#) on page 1-6. The Component Maker window opens. In the Components pane (Figure 3-1 on page 3-6), double-click Structure-Based.
- 2 The view shown in Figure 3-2 on page 3-7 opens. This view shows the structure-based logical components created for the programs in the current project.

**3-12** Extracting Structure-Based Components  
*Extracting Structure-Based PL/I Components (Future Release)*

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


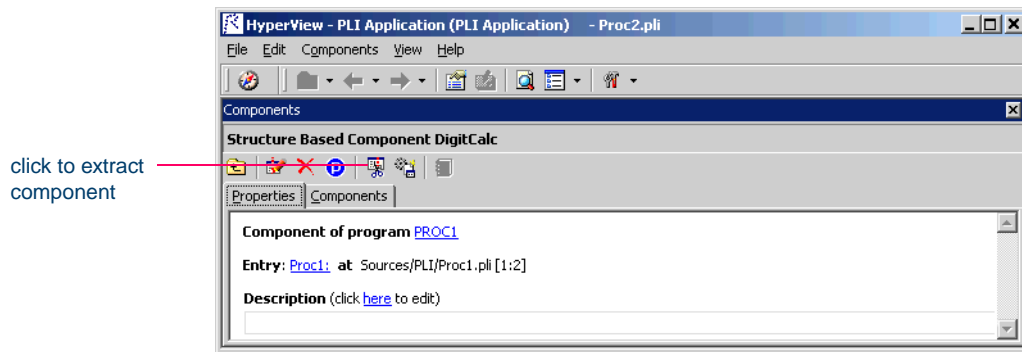
- 3A** To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B** To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4** The view shown in Figure 3-7 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 3-7 *Components Pane — Properties Tab (PL/I)*



- 5** Select the program entry point in the Source pane. In the **Entry** field, click the link for the current selection and choose **Set** in the pop-up menu.

**Note:** To unset an entry point, click it and choose **Unset** in the pop-up menu. To navigate quickly to an entry point in the source, click it and choose **Locate** in the pop-up menu.

- 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.


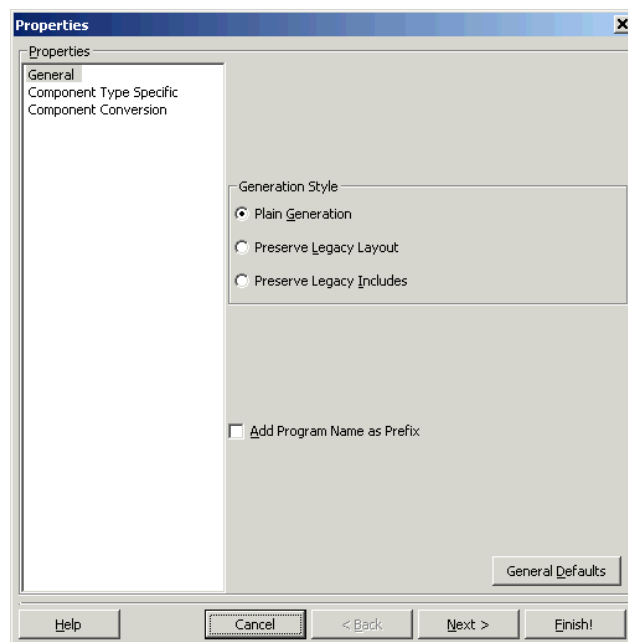
- 7 Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
- 8 The dialog shown in Figure 3-8 opens. This dialog displays a series of panes that let you set extraction options for Structure-Based Componentization. For usage information, see [“Setting PL/I Extraction Options” on page 2-13](#). When you are satisfied with your choices, click **Finish**.

Figure 3-8 *Components Pane — Extraction Options Dialog (PL/I)*



- 9 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**.

**3-14** Extracting Structure-Based Components  
*Extracting Structure-Based Natural Components (Future Release)*


- 10 Assuming the extraction executed without errors, the view shown in Figure 3-6 on page 3-11 opens. This view shows a list of the component source files that were generated for the logical component. Click an item in the list to view the read-only text for the item.

## ***Extracting Structure-Based Natural Components (Future Release)***

This section describes how to perform Structure-Based Componentization for Natural programs.

***To extract a structure-based Natural component:***

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 3-1 on page 3-6), double-click Structure-Based.
- 2 The view shown in Figure 3-2 on page 3-7 opens. This view shows the structure-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.


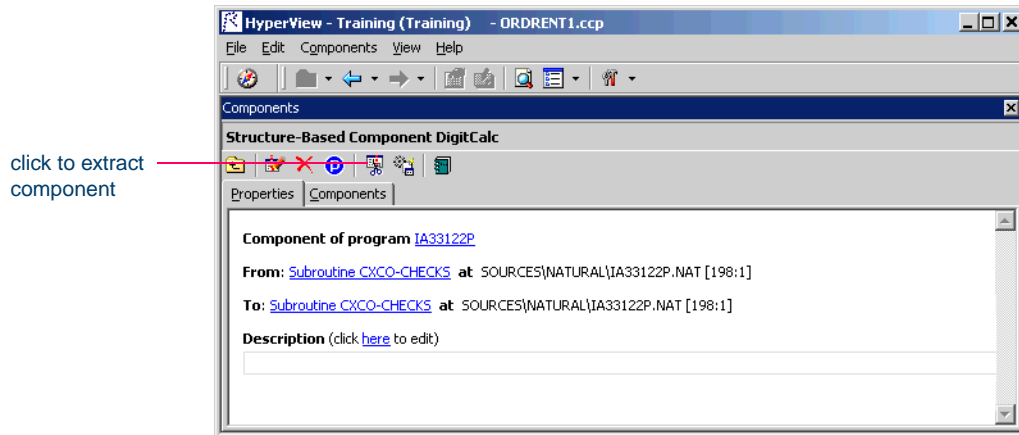

- 3A To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 3-9 on page 3-15 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 3-9 *Components Pane — Properties Tab (Natural)*



- 5 Select the first subroutine in the range in the Source pane, then click the link for the current selection in the **From** field and choose **Set** in the drop-down menu. Select the second subroutine in the pair, then click the link for the current selection in the **To** field and choose **Set** in the pop-up menu. For background, see [“Understanding Ranges” on page 3-1](#).

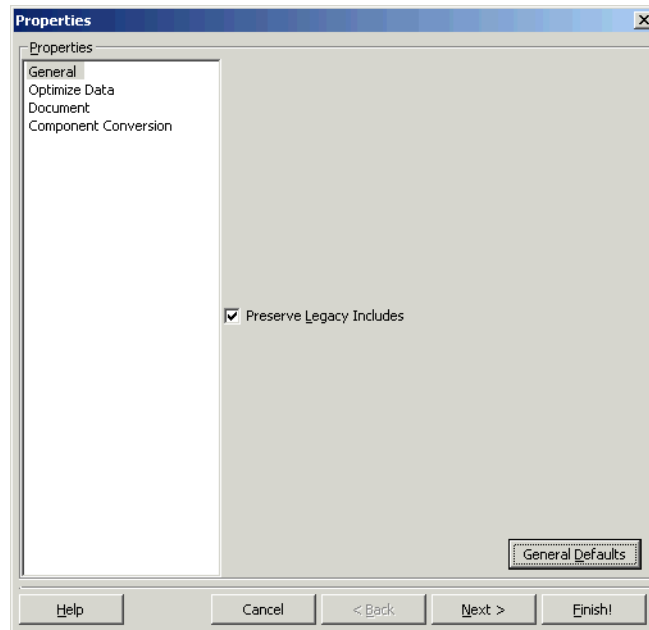
**Note:** To unset a subroutine, click it and choose **Unset** in the pop-up menu. To navigate quickly to a subroutine in the source, click it and choose **Locate** in the pop-up menu.

- 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 start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
- 8 The dialog shown in Figure 3-10 on page 3-16 opens. This dialog displays a series of panes that let you set extraction options for Structure-Based Componentization. For usage information, see [“Setting](#)

**3-16** Extracting Structure-Based Components  
*Extracting Structure-Based Natural Components (Future Release)*

Natural Extraction Options” on page 2-15. When you are satisfied with your choices, click **Finish**.

Figure 3-10 *Components Pane — Extraction Options Dialog (Natural)*




- 9 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**.
- 10 Assuming the extraction executed without errors, the view shown in Figure 3-6 on page 3-11 opens. This view shows a list of the component source files that were generated for the logical component. Click an item in the list to view the read-only text for the item.


## **Extracting External Subroutines from Natural Programs (Future Release)**

For Natural programs, you can use a separate method called *External Subroutine Componentization* to extract a single internal subroutine as an external subroutine.

### **To extract an external subroutine:**

- 1** Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 3-1 on page 3-6), double-click External Subroutine.
- 2** The view shown in Figure 3-2 on page 3-7 opens. This view shows the external subroutine 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.

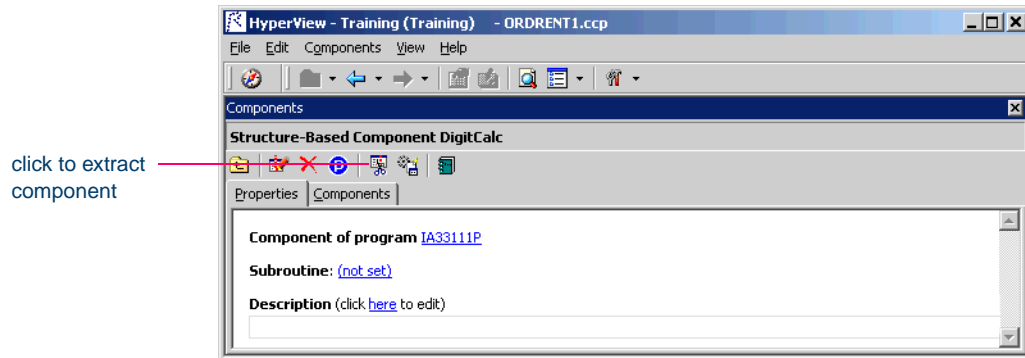
- 3A** To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B** To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4** The view shown in Figure 3-11 on page 3-18 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).
- 5** Select the internal subroutine in the Source pane. In the **Subroutine** field, click the link for the current selection and choose **Set** in the pop-up menu.

**Note:** To unset a subroutine, click it and choose **Unset** in the pop-up menu. To navigate quickly to a subroutine in the source, click it and choose **Locate** in the pop-up menu.

**3-18** Extracting Structure-Based Components  
*What's Next?*

- 6 Follow steps [6-10](#) on pages 3-15-3-16 to complete the extraction.

Figure 3-11 *Components Pane — Properties Tab (External Subroutine)*



## ***What's Next?***

Now that you know how to extract a component based on a range of inline code, let's look at how you use Component Maker to perform more complex extractions, starting with Computation-Based Componentization.



## *Extracting Computation-Based Components*



**C**omputation-Based Componentization lets you build a component that contains all the code necessary to calculate the value of a variable at a particular point in a program — the value of a DayOfTheWeek variable, for example, where it is used to populate a report attribute or screen. As with structure-based slices, you can generate parameterized computation-based slices that make it easy to deploy the transformed component in distributed architectures. For background, see [“Understanding Parameterized Slices” on page 3-3](#).

### ***Set Cobol Verification Options!***

For computation-based componentization of Cobol programs, you *must* turn on not only the **Perform Program Analysis** option in the Verification tab of the Project Options window, but also the **Enable Extraction of Computation-Based Components** option in the same dialog. For information on how to set verification options, see *Preparing Projects* in the ATW document set.

## ***Understanding Variable-Based Extraction***

When you perform a computation-based extraction, you can slice by *statement* or by *variable*. What's the difference? Suppose you are interested in calculations involving the variable X in the example below:

```
MOVE 1 TO X  
MOVE 1 TO Y  
DISPLAY X Y.
```

If you perform statement-based extraction — if you slice on the statement DISPLAY X Y — all three statements will be included in the component. If you perform variable-based extraction — if you slice on the variable X — only the first and third statements will be included. In variable-based extraction, that is, Component Maker tracks the dependency between X and Y, and having determined that the variables are independent, excludes the MOVE 1 to Y statement.

**Note:** If you slice on a variable for a Cobol component, you must select **Variable** in the Component Type Specific options for computation-based extraction. For more information, see [step 2 on page 2-9](#).

## ***Understanding Blocking***

For Cobol programs, you can use a technique called *blocking* to produce smaller, better-defined parameterized components. Component Maker will not include in the slice any part of the calculation that appears before the blocked statement. Fields from blocked input statements are treated as input parameters of the component.

Consider the following fragment:

```
INP1.  
  DISPLAY "INPUT YEAR (1600-2099)".  
  ACCEPT YEAR.
```

```
CALL 'PROG' USING YEAR.  
IF YEAR > 2099 OR YEAR < 1600 THEN  
  DISPLAY "WRONG YEAR".
```

If the CALL statement is selected as a block, then both the CALL and ACCEPT statements from the fragment are not included in the component, and YEAR is passed as a parameter to the component.

#### ***How to Specify Blocking***

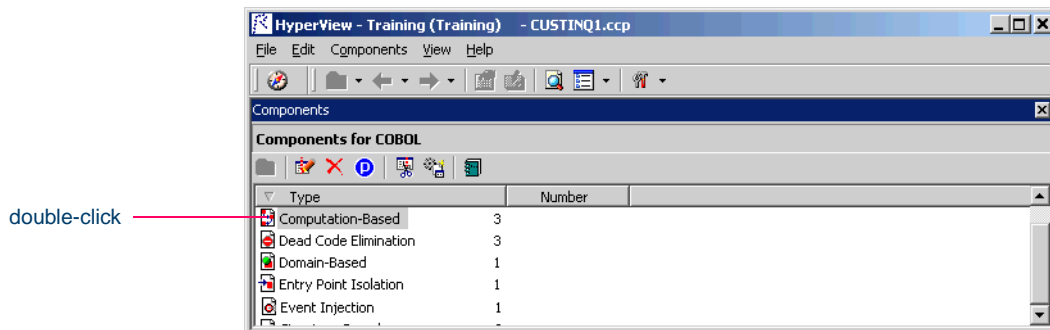
Specify blocking in the blocking dialog (Figure 2-1 on page 2-4), accessed from the Interface options pane. For information on how to specify blocking, see [step 4 on page 2-4](#).

## ***Extracting Computation-Based Cobol Components***

This section describes how to perform Computation-Based Componentization for Cobol programs.

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 4-1), double-click Computation-Based.

Figure 4-1 *Components Pane*



- 2 The view shown in Figure 4-2 on page 4-4 opens. This view shows the computation-based logical components created for the programs in the current project.

4-4 Extracting Computation-Based Components  
*Extracting Computation-Based Cobol Components*


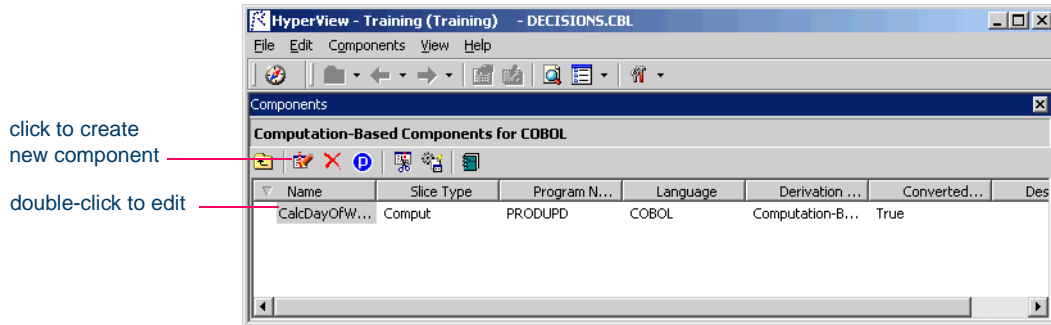
**Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.

Figure 4-2 *Components Pane — Computation-Based View*




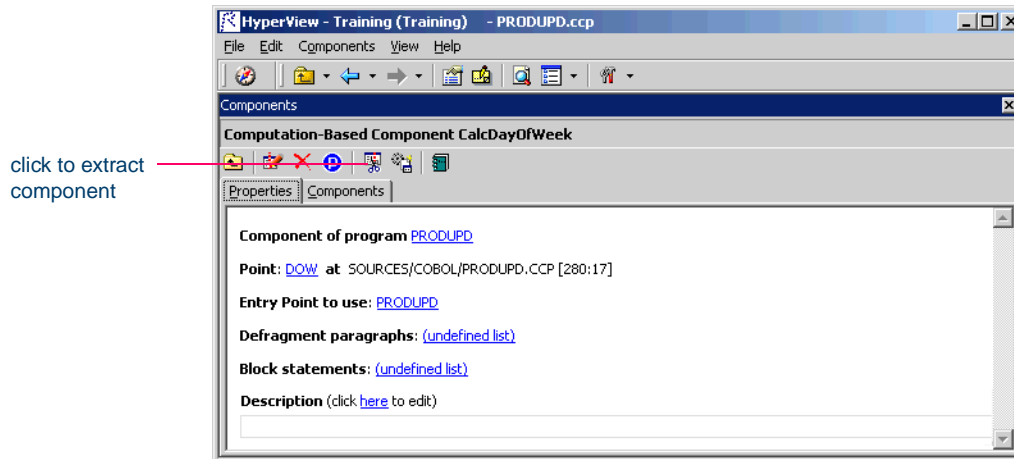
- 3A To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 4-3 on page 4-5 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 4-3 Components Pane — Properties Tab (Cobol)



- 5 Select the variable or statement you want to slice on in the Source pane. In the **Point** field, click the link for the current selection and choose **Set** in the pop-up menu. To unset a variable or statement, click it and choose **Unset** in the pop-up menu. To navigate quickly to a variable or statement in the source, click it and choose **Locate** in the pop-up menu.

**Note:** If you slice on a variable, you must select **Variable** in the Component Type Specific options for computation-based extraction. For more information, see [step 2 on page 2-9](#).

- 6 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. For background, see Chapter 8, "Performing Entry Point Isolation." To unset an entry point, click it and choose **Unset** in the pop-up menu.
- 7 In the **Defragment paragraphs** field, click the link for the current selection and choose the list of paragraphs to defragment in the pop-up menu. For background, see "[Understanding Refactoring \(Future Release\)](#)" on page 7-3.

**Note:** To create a new list of paragraphs to defragment, choose **New** in the pop-up menu. The List Browser opens, where you can

4-6 Extracting Computation-Based Components  
*Extracting Computation-Based Cobol Components*

create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.


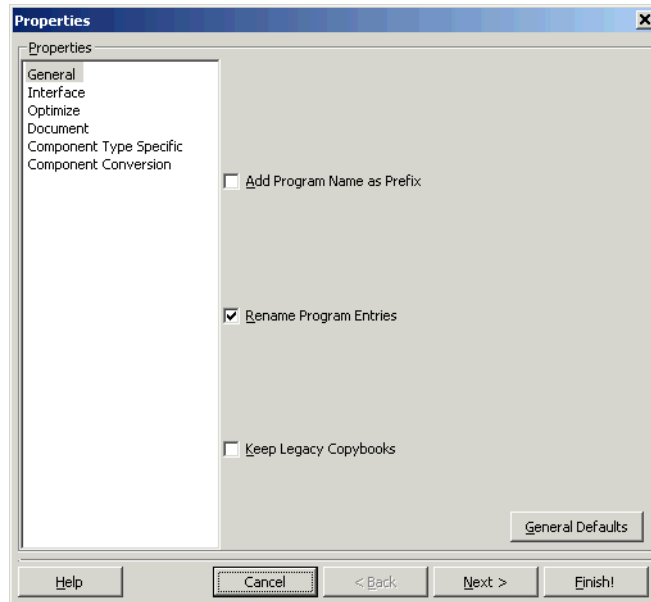
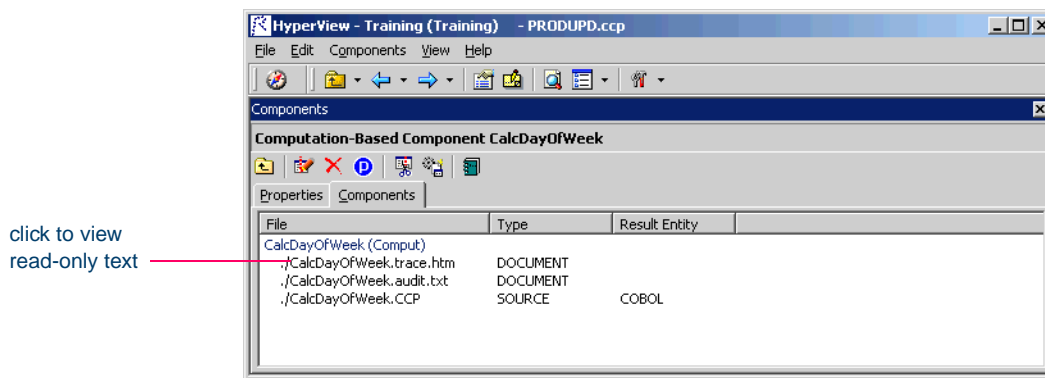
- 8 In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
- 9 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**.
- 10 The dialog shown in Figure 4-4 opens. This dialog displays a series of panes that let you set extraction options for Computation-Based Componentization. For usage information, see [“Setting Cobol Extraction Options” on page 2-2](#). When you are satisfied with your choices, click **Finish**.

Figure 4-4 Components Pane — Extraction Options Dialog (Cobol)



- 11 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**.
- 12 Assuming the extraction executed without errors, the view shown in Figure 4-5 opens. This view shows a list of the component source files that were generated for the logical component and an audit report if you requested one (see [“Generating Audit Reports \(Cobol Only\)”](#) on page 1-13). Click an item in the list to view the read-only text for the item.

Figure 4-5 *Components Pane — Components Tab*




## ***Extracting Computation-Based Natural Components***

This section describes how to perform Computation-Based Componentization for Natural programs.

### ***To extract a computation-based Natural component:***

- 1 Start Component Maker as described in [“Starting Component Maker”](#) on page 1-6. The Component Maker window opens. In the Components pane (Figure 4-1 on page 4-3), double-click Computation-Based.

- 2 The view shown in Figure 4-2 on page 4-4 opens. This view shows the computation-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.


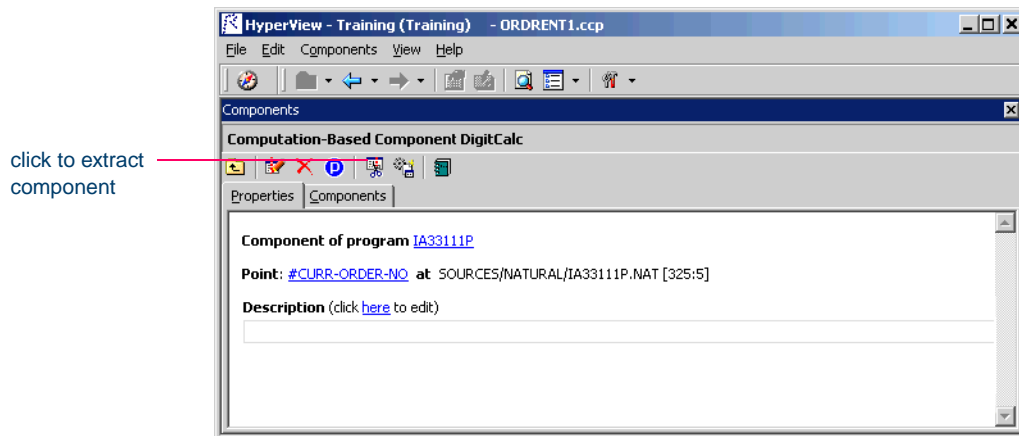
- 3A To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 4-6 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 4-6 *Components Pane — Properties Tab (Natural)*



- 5 Select the variable or statement you want to slice on in the Source pane. In the **Point** field, click the link for the current selection and choose **Set** in the pop-up menu. To unset a variable or statement, click it and choose **Unset** in the pop-up menu. To navigate quickly to a variable or statement in the source, click it and choose **Locate** in the pop-up menu.




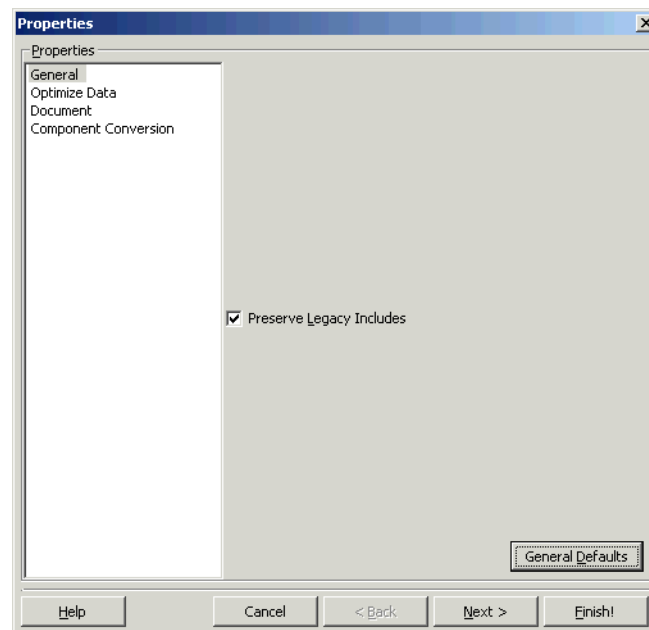
- 6 In the **Description** field, click the [here](#) link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
- 7 Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
- 8 The dialog shown in Figure 4-7 opens. This dialog displays a series of panes that let you set extraction options for Computation-Based Componentization. For usage information, see [“Setting Natural Extraction Options” on page 2-15](#). When you are satisfied with your choices, click **Finish**.

Figure 4-7 *Components Pane — Extraction Options Dialog (Natural)*



**4-10** Extracting Computation-Based Components  
*What's Next?*

- 9 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**.
- 10 Assuming the extraction executed without errors, the view shown in Figure 4-5 on page 4-7 opens. This view shows a list of the component source files that were generated for the logical component. Click an item in the list to view the read-only text for the item.

### ***What's Next?***

That completes our survey of Computation-Based Componentization. Now let's look at how you use Component Maker to perform Domain-Based Componentization. That's the subject of the next chapter.

## *Extracting Domain-Based Components*



**D**omain-Based Componentization lets you “specialize” a program based on the values of one or more variables. The specialized program is typically intended for reuse “in place” — in the original application but under new external circumstances.

After a change in your business practices, for example, a program that invokes processing for a “payment type” variable could be specialized on the value `PAYMENT-TYPE = "CHECK"`. Component Maker isolates every process dependent on the `CHECK` value to create a functionally complete program that processes check payments only.

Two modes of domain-based componentization are offered:

- The *simplified* mode treats the variable as a constant throughout the execution of the program — the value of the variable is “frozen in memory.” Operations that could change the value are ignored.
- The *advanced* mode sets the value of the variable only at a specified data port — subsequent operations can change the value, following the data and control flow of the program.

**5-2** Extracting Domain-Based Components  
*Understanding Program Specialization in Simplified Mode*

Use the simplified mode when you are interested only in the final value of a variable, or when a variable never receives a value from outside the program. Use the advanced mode when you need to account for data coming into a variable — when the variable’s value is repeatedly reset, for example. The next two sections describe these modes in detail.

**Setting a Specialization Variable to Multiple Values**

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.

**Understanding Program Specialization in Simplified Mode**

The simplified mode of program specialization treats the specialization variable as a constant throughout the execution of the program. Operations that could change the value are ignored.

Table 5-1 shows the result of using the simplified mode to specialize on the values CURYEAR = 1999, MONTH = 1, CURMONTH = 12, DAY1 = 4, and CURDAY = 7.

Table 5-1 *Example of Program Specialization in Simplified Mode*

Source Program	Specialized Program	Comment
<pre> INP3.   DISPLAY "INPUT DAY".   ACCEPT DAY1.    MOVE YEAR TO tmp1.   PERFORM ISV.    IF DAY1 &gt; tt of MONTHS     ( MONTH ) OR DAY1 &lt; 1   THEN     DISPLAY "WRONG DAY". </pre>	<pre> INP3.   DISPLAY "INPUT DAY".    MOVE YEAR TO tmp1.   PERFORM ISV.    IF 0004 &gt; TT OF     MONTHS( MONTH ) THEN     DISPLAY "WRONG DAY"   END-IF. </pre>	<p>ACCEPT removed.</p> <p>No changes in these statements (YEAR is a “free” variable).</p> <p>Value for DAY1 substituted. The 2nd condition for DAY1 is removed as always false. END-IF added.</p>

Table 5-1 *Example of Program Specialization in Simplified Mode (continued)*

Source Program	Specialized Program	Comment
<pre> MAINCALC.   IF YEAR &gt; CURYEAR   THEN     MOVE YEAR TO       INT0001     MOVE CURYEAR TO       INT0002     MOVE 1 TO direction   ELSE     MOVE YEAR TO       INT0002     MOVE 2 TO direction     MOVE CURYEAR TO       INT0001. </pre>	<pre> MAINCALC.   IF YEAR &gt; 1999   THEN     MOVE YEAR TO INT0001     MOVE 1999 TO       INT0002     MOVE 1 TO direction   ELSE     MOVE YEAR TO INT0002     MOVE 2 TO direction     MOVE 1999 TO       INT0001. </pre>	Value for CURYEAR substituted.
<pre> MOVE int0001 TO   tmp3. MOVE int0002 TO   tmp4. IF YEAR NOT EQUAL   CURYEAR THEN   PERFORM YEARS. </pre>	<pre> MOVE int0002 TO tmp4. IF YEAR NOT = 1999 THEN   PERFORM YEARS. </pre>	Component Maker removes the first line for tmp3, because this variable is never used again. Value for CURYEAR substituted.
<pre> IF MONTH &gt; CURMONTH   THEN     MOVE MONTH TO       INT0001     MOVE CURMONTH TO       INT0002     MOVE 1 TO direction </pre>		Value for MONTH substituted, making the condition (1>12) false, so Component Maker removes the IF branch and then the whole conditional statement as such.
<pre> ELSE   MOVE MONTH TO     INT0002   MOVE 2 TO direction   MOVE CURMONTH TO     INT0001. </pre>	<pre> MOVE 0001 TO   INT0002 MOVE 2 TO direction MOVE 0012 TO   INT0001. </pre>	The three unconditional statements remain from the former ELSE branch. Value for CURMONTH substituted.

**5-4** Extracting Domain-Based Components  
*Understanding Program Specialization in Simplified Mode*

Table 5-1 *Example of Program Specialization in Simplified Mode (continued)*

Source Program	Specialized Program	Comment
<pre> IF MONTH NOT EQUAL CURMONTH THEN PERFORM MONTHS. </pre>	<pre> PERFORM MONTHS. </pre>	The condition is true, so the statement is made unconditional.
<pre> IF DAY1 &gt; CURDAY THEN MOVE DAY1 TO INT0001 MOVE CURDAY TO INT0002 MOVE 1 TO direction </pre>		This condition (4>7) is false, so Component Maker removes the IF branch and then the whole conditional statement as such.
<pre> ELSE MOVE DAY1 TO INT0002 MOVE 2 TO direction MOVE CURDAY TO INT0001. </pre>	<pre> MOVE 4 TO INT0002 MOVE 2 TO direction MOVE 0007 TO INT0001. </pre>	The three unconditional statements remain from the former ELSE branch. Values for DAY1 and CURDAY substituted.
<pre> IF day1 NOT EQUAL CURDAY THEN PERFORM DAYS. </pre>	<pre> PERFORM DAYS. </pre>	The condition is true, so the statement is made unconditional.

## ***Understanding Program Specialization in Advanced Mode***

The advanced mode of program specialization sets the value of the specialization variable only at a specified data port — at any statement that allows the program to receive the variable’s value from a keyboard, database, screen, or other input source. Subsequent operations can change the value, following the data and control flow of the program.

Table 5-2 on page 5-5 shows the result of using the advanced mode to specialize on the values MONTH = 1 and DAY1 = 4.

**Table 5-2**     *Example of Program Specialization in Advanced Mode*

<b>Source Program</b>	<b>Specialized Program</b>	<b>Comment</b>
<pre>INP1.   DISPLAY "INPUT YEAR     (1600-2099)".   ACCEPT YEAR.   IF YEAR &gt; 2099 OR YEAR     &lt; 1600 THEN     DISPLAY "WRONG YEAR".</pre>	<pre>INP1.   DISPLAY "INPUT YEAR     (1600-2099)".   ACCEPT YEAR.   IF YEAR &gt; 2099 OR YEAR     &lt; 1600 THEN     DISPLAY "WRONG YEAR".</pre>	<p>No changes in these statements (YEAR is a “free” variable).</p>
<pre>INP2.   DISPLAY "INPUT     MONTH".   ACCEPT MONTH.    IF MONTH &gt; 12 OR MONTH     &lt; 1 THEN     DISPLAY "WRONG MONTH".</pre>	<pre>INP2.   DISPLAY "INPUT MONTH".   MOVE 0001 TO MONTH.</pre>	<p>ACCEPT is replaced by MOVE with the set value for MONTH.</p> <p>With the set value, this IF statement can never be reached, so Component Maker removes it.</p>

5-6 Extracting Domain-Based Components  
*Understanding Program Specialization “Lite”*

Table 5-2 *Example of Program Specialization in Advanced Mode*

Source Program	Specialized Program	Comment
<pre>INP3.   DISPLAY "INPUT DAY".   ACCEPT DAY1.    MOVE YEAR TO tmp1.   PERFORM ISV.    IF DAY1 &gt; tt of MONTHS     ( MONTH ) OR DAY1 &lt; 1   THEN     DISPLAY "WRONG DAY".</pre>	<pre>INP3.   DISPLAY "INPUT DAY".   MOVE 0004 TO DAY1.    MOVE YEAR TO tmp1.   PERFORM ISV.    IF 0004 &gt; TT OF     MONTHS( MONTH ) THEN     DISPLAY "WRONG DAY"   END-IF.</pre>	<p>ACCEPT is replaced by MOVE with the set value for DAY1.</p> <p>No changes in these statements (YEAR is a “free” variable).</p> <p>The 2nd condition for DAY1 is removed as always false. END-IF added.</p>

### ***Understanding Program Specialization “Lite”***

Ordinarily, you must turn on the **Perform Program Analysis** option in the Verification tab of the Project Options window before verifying the Cobol program you want to specialize. If your application is very large, however, *and* you know that the specialization variable is never reset, you can save time by skipping program analysis during verification and using the simplified mode to specialize the program — so-called “program specialization lite.”

Component Maker gives you the same result for a lite extraction as it would for an ordinary domain extraction in simplified mode, with one important exception. Domain extraction lite cannot calculate the value of a variable that depends on the value of the specialization variable. Consider the following example:

```
01 X Pic 99.
01 Y Pic 99.
...
MOVE X To Y.
IF X = 1
  THEN ...
  ELSE ...
```



```
END-IF .  
...  
IF Y = 1  
    THEN ...  
    ELSE ...  
END-IF .
```

If you set X to 1, both simplified mode and domain extraction lite resolve the IF X = 1 condition correctly. Only simplified mode, however, resolves the IF Y = 1 condition.

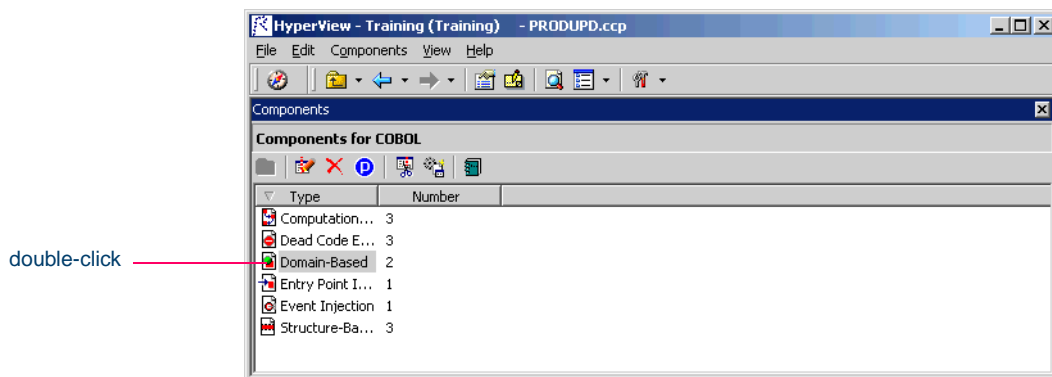
## Extracting Domain-Based Cobol Components

This section describes how to perform Domain-Based Componentization for Cobol programs.

### To extract a domain-based Cobol component:

- 1 Start Component Maker as described in “[Starting Component Maker](#)” on page 1-6. The Component Maker window opens. In the Components pane (Figure 5-1), double-click Domain-Based.

Figure 5-1 Components Pane



- 2 The view shown in Figure 5-2 on page 5-8 opens. This view shows the domain-based logical components created for the programs in the current project.

5-8 Extracting Domain-Based Components  
*Extracting Domain-Based Cobol Components*


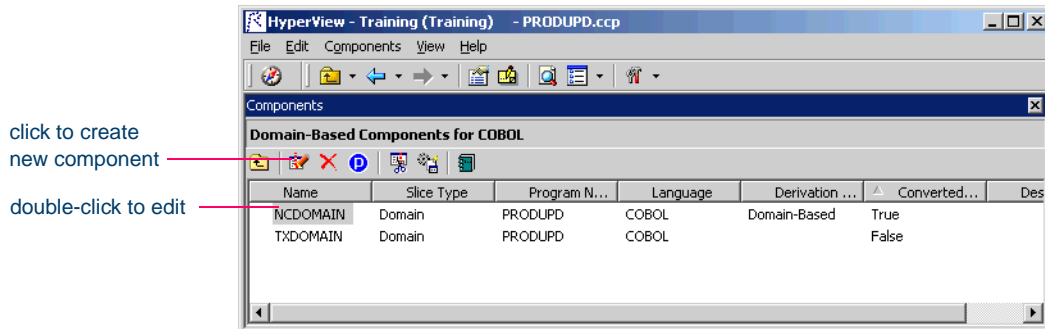
**Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.

Figure 5-2 *Components Pane — Domain-Based View*




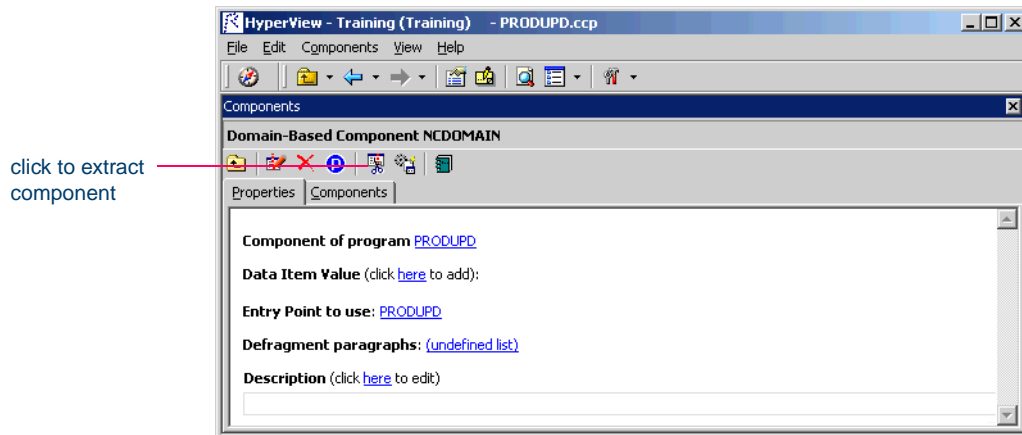
- 3A To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 5-3 on page 5-9 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 5-3 Components Pane — Properties Tab (Cobol)



- 5 In the **Data Item Value** field, click the **here** link. Choose one of the following methods in the pop-up menu.
    - **HyperCode List** to set the specialization variable to the constant values in the list of constants specified in [step 8A on page 5-10](#).
    - **User Specified Value(s)** to set the specialization variable to the value or values specified in [step 8B on page 5-10](#).
  - 6A To use the simplified mode of program specialization (see [“Understanding Program Specialization in Simplified Mode” on page 5-2](#)), select the declaration for the specialization variable in the Source pane, then click the link for the current selection in the **Data Item** field and choose **Set** in the drop-down menu.
  - 6B To use the advanced mode of program specialization (see [“Understanding Program Specialization in Advanced Mode” on page 5-5](#)), select the data port of interest for the specialization variable in the Source pane, then click the link for the current selection in the **Data Item** field and choose **Set** in the drop-down menu.
- Note:** To delete a declaration or data port, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu. To unset a declaration or data port, click it and choose **Unset**

in the pop-up menu. To navigate quickly to a declaration or data port in the source, click it and choose **Locate** in the pop-up menu.

- 7 In the **Comparison** field, click the link for the current comparison operator and choose:
- **equals** to set the specialization variable to the values specified in [step 8A](#) or [step 8B](#).
  - **not equals** to set the specialization variable to every value *but* the values specified in [step 8A](#) or [step 8B](#).

- 8A If you chose **HyperCode List** in [step 5 on page 5-9](#), click the link for the current selection in the **List Name** field and choose the list of constants to use in the pop-up menu.

**Note:** To create a new list of constants, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.

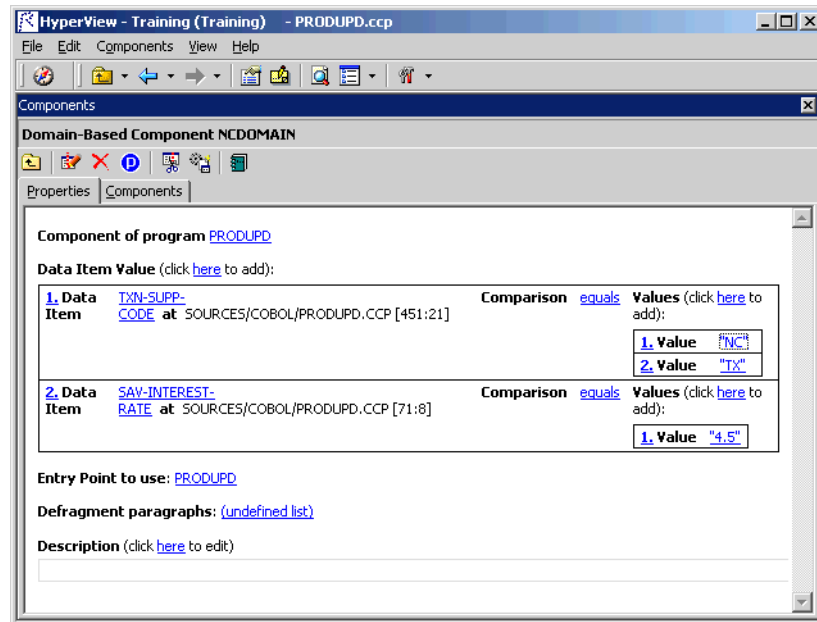
- 8B If you chose **User Specified Value(s)** in [step 5 on page 5-9](#), click the **here** link in the **Values** field. Choose one of the following methods in the pop-up menu. For background on the methods, see [“Setting a Specialization Variable to Multiple Values” on page 5-2](#).

- **Value** to set the specialization variable to one or more values. In the **Value** field, click the link for the current selection. A dialog opens where you can enter a value in the text field. Click **OK**.
- **Value Range** to set the specialization variable to a range of values. In the **Lower** field, click the link for the current selection. A dialog opens where you can enter a value for the lower range end in the text field. Click **OK**. Follow the same procedure for the **Upper** field.

**Note:** Put double quotation marks around a string constant with blank spaces at the beginning or end.

- 9 Repeat [step 8A](#) or [step 8B](#) for each list, value, or range of values you want to set. For a given specialization variable, you can specify the methods in any combination. To delete a list, value, or range, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu.
- 10 Repeat steps [5-9](#) for each variable you want to specialize on. For a given extraction, you can specify simplified and advanced modes in any combination. Figure 5-4 shows how the properties tab might look for a multi-variable extraction.

Figure 5-4 *Properties Tab — Multi-Variable Cobol Extraction*



- 11 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. For background, see [Chapter 8, "Performing Entry Point Isolation."](#) To unset an entry point, click it and choose **Unset** in the pop-up menu.
- 12 In the **Defragment paragraphs** field, click the link for the current selection and choose the list of paragraphs to defragment in the pop-

up menu. For background, see [“Understanding Refactoring \(Future Release\)” on page 7-3](#).


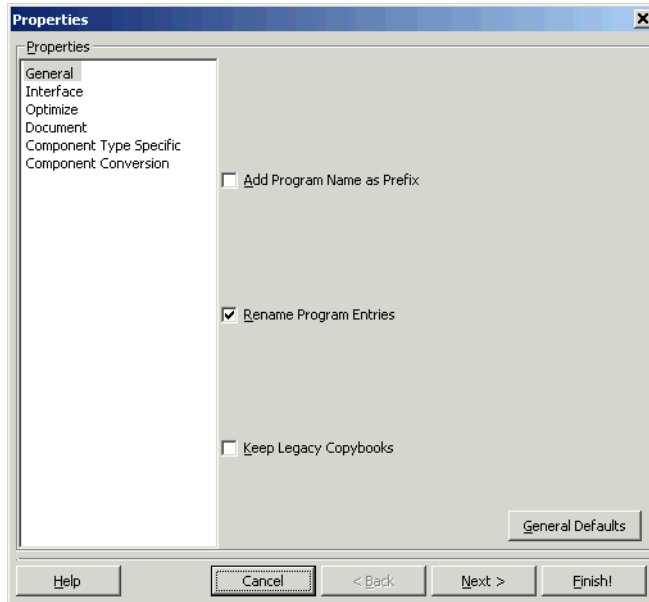
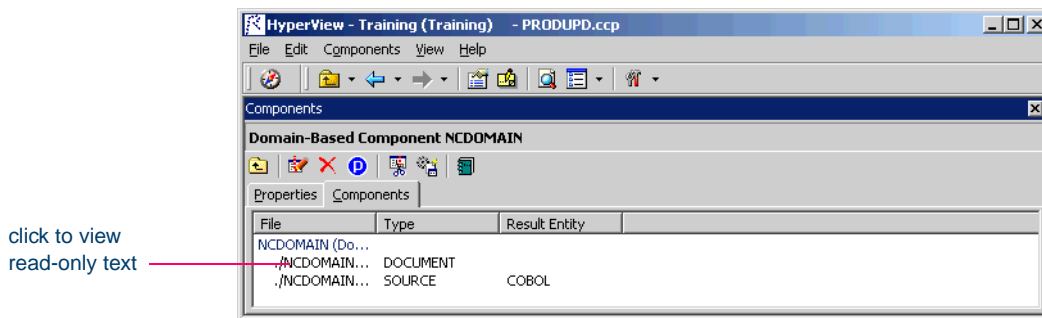
- Note:** To create a new list of paragraphs to defragment, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.
- 13 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.
  - 14 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**.
  - 15 The dialog shown in Figure 5-5 on page 5-13 opens. This dialog displays a series of panes that let you set extraction options for Domain-Based Componentization. For usage information, see [“Setting Cobol Extraction Options” on page 2-2](#). When you are satisfied with your choices, click **Finish**.
  - 16 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**.

Figure 5-5 *Components Pane — Extraction Options Dialog (Cobol)*



- 17 Assuming the extraction executed without errors, the view shown in Figure 5-6 opens. This view shows a list of the component source files that were generated for the logical component and an audit report if you requested one (see [“Generating Audit Reports \(Cobol Only\)”](#) on page 1-13). Click an item in the list to view the read-only text for the item.

Figure 5-6 *Components Pane — Components Tab*




## **Extracting Domain-Based PL/I Components**

This section describes how to perform Domain-Based Componentization for PL/I programs.

**Note:** Not-equals comparisons and value ranges are not supported in PL/I.

### **To extract a domain-based PL/I component:**

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 5-1 on page 5-7), double-click Domain-Based.
- 2 The view shown in Figure 5-2 on page 5-8 opens. This view shows the domain-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.


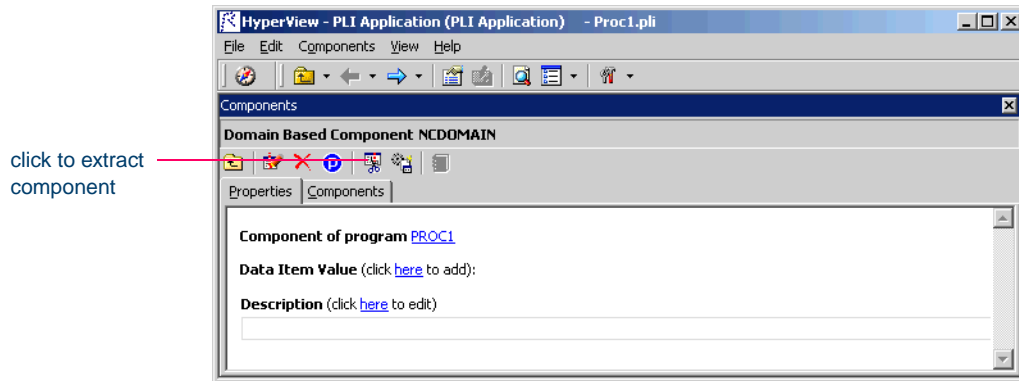
- 3A To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 5-7 on page 5-15 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).



Figure 5-7 Components Pane — Properties Tab (PL/I)



- 5 In the **Data Item Value** field, click the **here** link. Choose one of the following methods in the pop-up menu.
  - **HyperCode List** to set the specialization variable to the constant values in the list of constants specified in [step 7 on page 5-16](#).
  - **User Specified Value(s)** to set the specialization variable to the value or values specified in [step 8 on page 5-16](#).
- 6A To use the simplified mode of program specialization (see [“Understanding Program Specialization in Simplified Mode” on page 5-2](#)), select the declaration for the specialization variable in the Source pane, then click the link for the current selection in the **Data Item** field and choose **Set** in the drop-down menu.
- 6B To use the advanced mode of program specialization (see [“Understanding Program Specialization in Advanced Mode” on page 5-5](#)), select the data port of interest for the specialization variable in the Source pane, then click the link for the current selection in the **Data Item** field and choose **Set** in the drop-down menu.


**Note:** To delete a declaration or data port, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu. To unset a declaration or data port, click it and choose **Unset** in the pop-up menu. To navigate quickly to a declaration or data port in the source, click it and choose **Locate** in the pop-up menu.

- 7 If you chose **HyperCode List** in [step 5 on page 5-15](#), click the link for the current selection in the **List Name** field and choose the list of constants to use in the pop-up menu.

**Note:** To create a new list of constants, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.

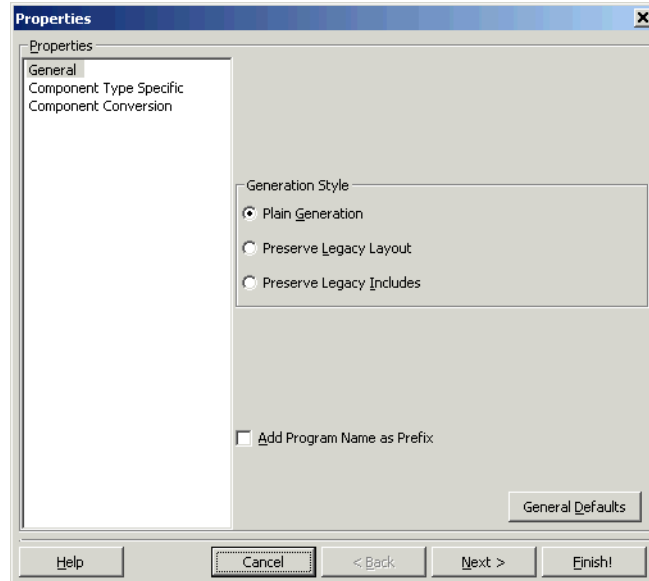
- 8 If you chose **User Specified Value(s)** in [step 5 on page 5-15](#), click the **here** link in the **Values** field. In the **Value** field, click the link for the current selection. A dialog opens where you can enter a value in the text field. Click **OK**.

**Note:** Put double quotation marks around a string constant with blank spaces at the beginning or end.

- 9 Repeat [step 7](#) or [step 8](#) for each list or value you want to set. For a given specialization variable, you can specify the methods in any combination. To delete a list or value, select the link for the numeral that identifies it and choose **Delete** in the pop-up menu.
- 10 Repeat steps [5-9](#) for each variable you want to specialize on. For a given extraction, you can specify simplified and advanced modes in any combination.
- 11 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.
- 12 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**.
- 13 The dialog shown in Figure 5-8 on page 5-17 opens. This dialog displays a series of panes that let you set extraction options for Domain-Based Componentization. For usage information, see [“Setting PL/I](#)

[Extraction Options](#)” on page 2-13. When you are satisfied with your choices, click **Finish**.

Figure 5-8 *Components Pane — Extraction Options Dialog (PL/I)*



- 14 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**.
- 15 Assuming the extraction executed without errors, the view shown in Figure 5-6 on page 5-13 opens. This view shows a list of the component source files that were generated for the logical component. Click an item in the list to view the read-only text for the item.

## **What's Next?**

Now that you know how to perform the basic componentization methods, let's look at how you use Component Maker to perform more specialized tasks. The next chapter looks at Event Injection.

**5-18** Extracting Domain-Based Components  
*What's Next?*

## *Injecting Events*



**E**vent Injection lets you adapt a legacy program to asynchronous, event-based programming models like MQ Series. You specify candidate locations for event calls — reads/writes, screen transactions, or subprogram calls, for example; the type of operation the event call performs — put or get; and the text of the message. For a put operation, for example, Component Maker builds a component that sends the message and any associated variable values to a queue, where the message can be retrieved by monitoring applications.

### ***Creating Candidate Lists in Clipper***

The HyperView Clipper pane lets you create lists of candidate locations for event injection. Use the predefined searches for file ports, screen ports, and subprogram calls, or define your own searches. For Clipper pane usage, see *Analyzing Programs* in the ATW document set.

## ***Understanding Event Injection***

Suppose that you have a piece of code that checks whether the variables YEAR and MONTH belong to admissible ranges:

```
IF YEAR > 2099 OR YEAR < 1600 THEN
  MOVE "WRONG YEAR" TO DOW1
ELSE
  IF MONTH > 12 OR MONTH < 1 THEN
    MOVE "WRONG MONTH" TO DOW1
  ELSE
    MOVE YEAR TO tmp1
    PERFORM ISV
```

Suppose further that you want to send a message to your MQ Series middleware each time valid dates are entered in these fields, along with the value that was entered for YEAR. Here, in schematic form, is the series of steps you would perform in Component Maker to accomplish these tasks.

- 1 In HyperView, create a list that contains the MOVE YEAR TO tmp1 statement in the List Browser.
- 2 In Component Maker, create a logical component with the following properties:
  - **Component of program** — select the program that contains the fragment.
  - **List** — select the list you created in [step 1](#).
  - **Insert** — specify where you want event-handling code to be injected, **before** or **after** the injection point. We'll inject event-handling code **after** the MOVE statement.
  - **Operation** — select the type of operation you want the event-handling code to perform, **put** or **get**. Since we want to send a message to middleware, we choose **put**.
  - **Include Values** — specify whether you want the values of variables at the injection point to be included with the generated mes-

sage. Since we want to send the value of YEAR with the message, we choose **true**.

- **Message** — specify the text of the message you want to send. In our case, the text is “Valid dates entered”.
- 3** In Component Maker, extract the logical component, making sure to set the **Use Middleware** drop-down in the Component Type Specific options for the extraction to MQ.

The result of the extraction appears below. Notice that Component Maker has arranged to inserted the text of the message and the value of the YEAR variable into the buffer, and inserted the appropriate PERFORM PUTQ statements into the code.

```

IF YEAR > 2099 OR YEAR < 1600 THEN
    MOVE "WRONG YEAR" TO DOW1
ELSE
IF MONTH > 12 OR MONTH < 1 THEN
    MOVE "WRONG MONTH" TO DOW1
ELSE
    MOVE '<TEXT Value= "Valid dates
        entered"></TEXT>' TO BUFFER
    PERFORM PUTQ
    STRING '<VAR Name= "YEAR" Value=
        "' YEAR "'></VAR>'
        '<VAR Name= "TMP1" Value= "' TMP1 "'></VAR>'
    DELIMITED BY SIZE
    INTO BUFFER END-STRING
    PERFORM PUTQ
    MOVE YEAR TO tmp1
    PERFORM ISV

```

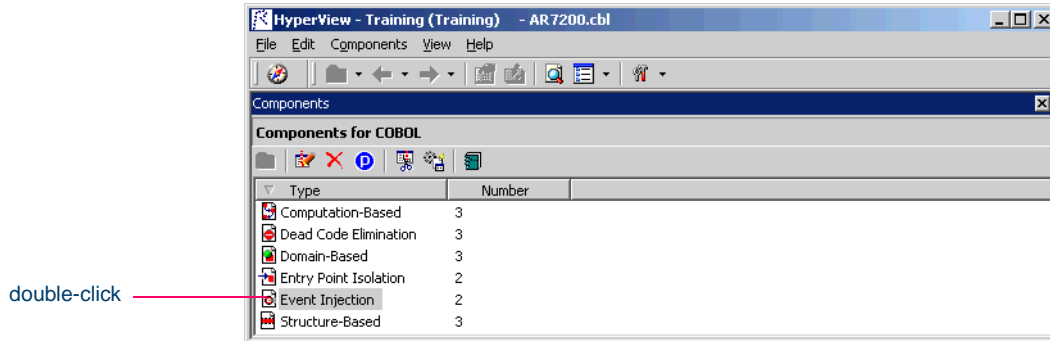
## Extracting Event-Injected Cobol Components

This section describes how to perform Event Injection for Cobol programs.

### To extract an event-injected Cobol component:

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 6-1), double-click Event Injection.

Figure 6-1 *Components Pane*



- 2 The view shown in Figure 6-2 on page 6-5 opens. This view shows the event-injected logical components created for the programs in the current project.


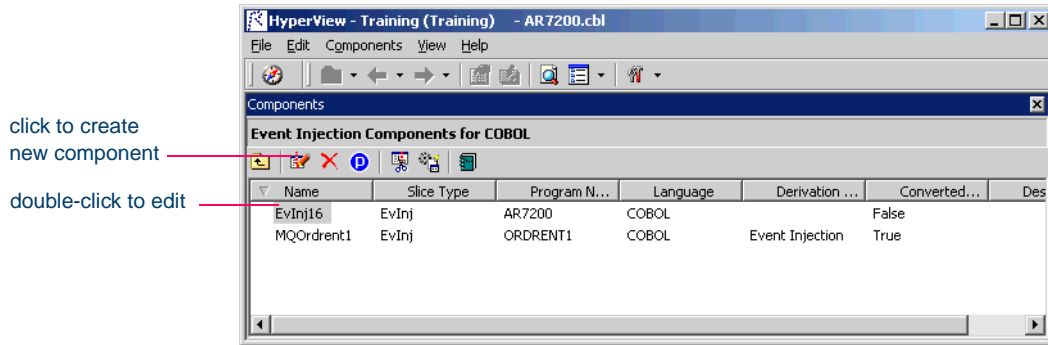
**Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.



Figure 6-2 *Components Pane — Event Injection View*




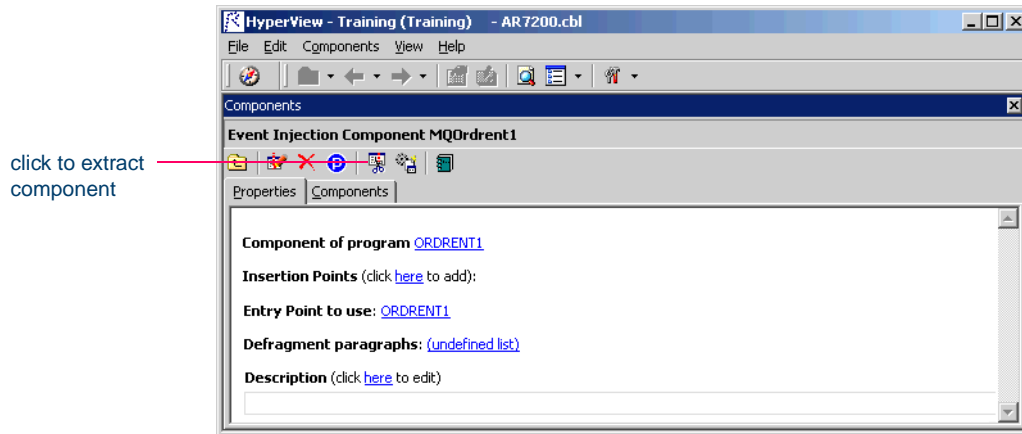
- 3A To create a new logical component, select the program you want to inject in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 6-3 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 6-3 *Components Pane — Properties Tab (Cobol)*



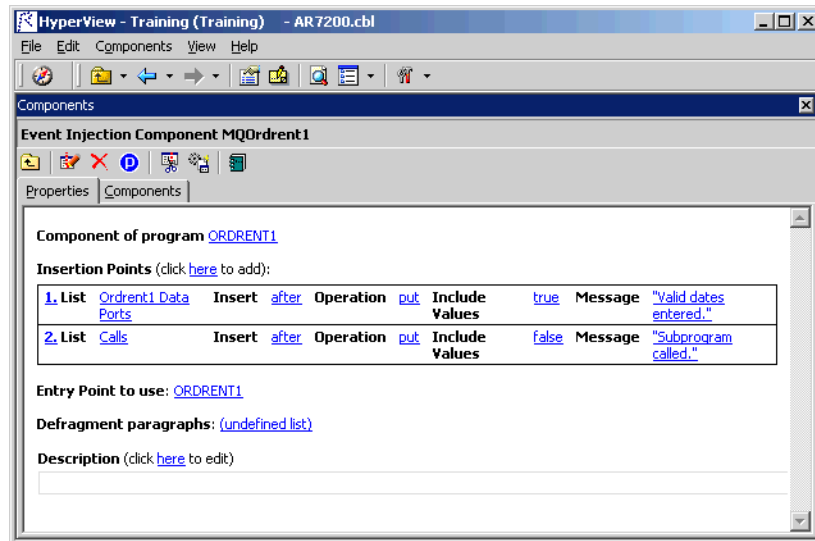
- 5 In the **Insertion Points** field, click the **here** link. In the **List** field, click the link for the current selection and choose the list of injection points in the pop-up menu. For background, see [“Understanding Event Injection”](#) on page 6-2.

**Note:** To delete a list, select the link for the numeral that identifies the list and choose **Delete** in the pop-up menu.

To create a new list of injection points, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set. See also [“Creating Candidate Lists in Clipper”](#) on page 6-1.

- 6 In the **Insert** field, click the link for the current selection and choose **after** to inject event-handling code after the selected injection point, **before** to inject event-handling code before the selected injection point.
- 7 In the **Operation** field, click the link for the current selection and choose **put** if you want event-handling code to send a message to middleware, **get** if you want event-handling code to receive a message from middleware.
- 8 In the **Include Values** field, click the link for the current selection and choose **true** if you want the values of variables at the injection point to be included with the generated message, **false** otherwise.
- 9 In the **Message** field, click the link for the current message. A dialog opens where you can enter the text for the event message in the text field. Click **OK**.
- 10 Repeat steps [5-9](#) for each list of candidate injection points. For a given extraction, you can specify the properties for the selected lists in any combination. Figure 6-4 shows how the properties tab might look for an extraction with multiple lists.

Figure 6-4 *Properties Tab — Multiple-List Cobol Extraction*



- 11 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. For background, see Chapter 8, “Performing Entry Point Isolation.” To unset an entry point, click it and choose **Unset** in the pop-up menu.
  - 12 In the **Defragment paragraphs** field, click the link for the current selection and choose the list of paragraphs to defragment in the pop-up menu. For background, see [“Understanding Refactoring \(Future Release\)” on page 7-3](#).
- Note:** To create a new list of paragraphs to defragment, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.
- 13 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.


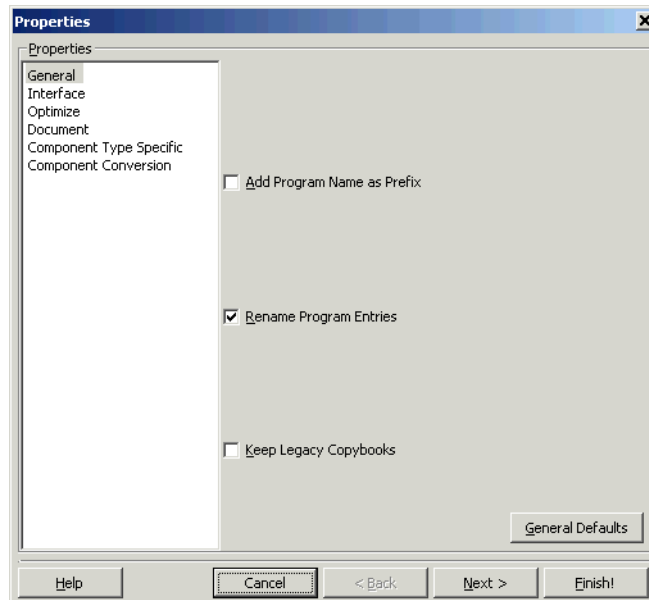
- 14 Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
- 15 The dialog shown in Figure 6-5 opens. This dialog displays a series of panes that let you set extraction options for Event Injection. For usage information, see [“Setting Cobol Extraction Options” on page 2-2](#). When you are satisfied with your choices, click **Finish**.

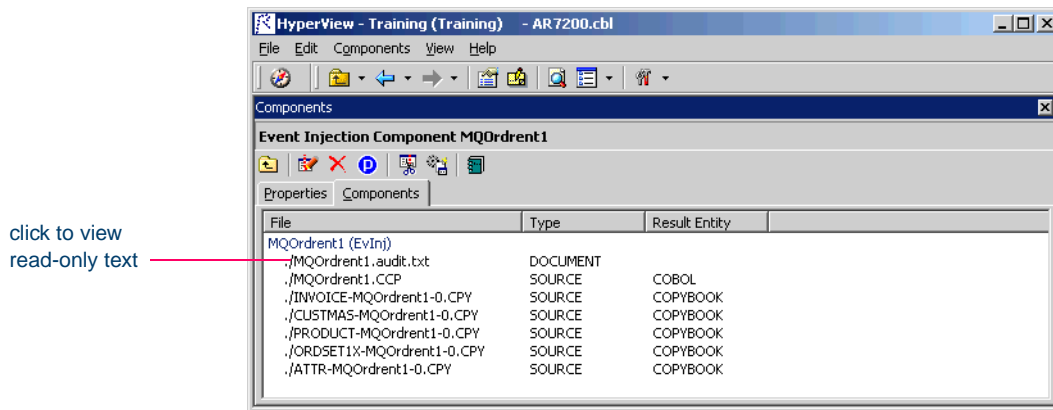
Figure 6-5 *Components Pane — Extraction Options Dialog (Cobol)*



- 16 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**.
- 17 Assuming the extraction executed without errors, the view shown in Figure 6-6 opens. This view shows a list of the component source

files that were generated for the logical component and an audit report if you requested one (see [“Generating Audit Reports \(Cobol Only\)” on page 1-13](#)). Click an item in the list to view the read-only text for the item.

Figure 6-6 Components Pane — Components Tab



## What's Next?

That completes our look at Event Injection. Now let's see how you use Component Maker to optimize programs.

**6-10** Injecting Events  
*What's Next?*

## *Eliminating Dead Code*



**D**ead 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, standalone DCE generates a component that consists of the original source code minus any unreferenced data items or unreachable procedural statements. For Cobol and Natural applications, set the options described in [Chapter 2](#) if you prefer to comment out dead code rather than remove it.

**Note:** You can perform Dead Code Elimination in batch mode, after batch registration and verification. For details, see *Preparing Projects* in the ATW document set.

### ***Refactoring Cobol Programs (Future Release)***

For Cobol programs, you can use an optimization method called *refactoring* to translate the program into a component with the same functionality and control flow but a simpler syntax structure. The component is said to be *defragmented*. For more information, see [“Understanding Refactoring \(Future Release\)” on page 7-3](#).

## Generating Dead Code Statistics

Set the **Perform Dead Code Analysis** option in the Verification tab of the Project Options window if you want the parser to collect statistics on the number of unreachable statements and dead data items in a program, and mark the constructs as dead in HyperView. You can view the statistics in the Legacy Estimation tool, as described in *Analyzing Projects* in the ATW document set.

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

### **Identifying Dead Code in Cobol Programs**

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. For usage details, see [“Generating Coverage Reports \(Cobol Only\)” on page 1-14](#).

## 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.  
MOVE 2 TO USED1.  
  
SECOND-USED PARA.  
MOVE 3 TO USED1.  
STOP RUN.
```

## ***Understanding Refactoring (Future Release)***

*Refactoring* lets you translate a Cobol program into a component with the same functionality and control flow but a simpler syntax structure. The component is said to be *defragmented*:

- It has fewer control transfers (GO TO and PERFORM statements).
- It has fewer linear sequences of operators. Sequentially executed statements are placed next to each other, so the program is less fragmented.

**7-4** Eliminating Dead Code  
*Understanding Refactoring (Future Release)*

Refactoring is an iterative process that identifies “basic blocks” in a program — sets of statements or paragraphs that have a continuous flow of control. It then unites the elements in each set and repeats the process until all possible blocks have been identified.

***Defragment Options***

Defragment options let you choose whether you want to defragment the whole program, only Component Maker-selected paragraphs, or only the paragraphs you have included in the “defragment list” specified in the Properties tab (see [step 6 on page 7-7](#)). For more information on the defragment options, see “[Optimize Options](#)” on page 2-4.

***Before:***

```
PROCEDURE DIVISION.  
  WBY21P00-MAIN SECTION.  
    PERFORM WBY21P10-INIT.  
    move 2 to i.  
    GO TO EZECONDLBL-00001.  
  EZECONDLBL-00001.  
    IF EZEAPP = " "  
      move 3 to i  
      GO TO EZECONDLBL-00002  
    END-IF  
    GO TO EZECONDLBL-00003  
    CONTINUE.  
  EZECONDLBL-00002.  
    move 4 to i  
    IF M1-FMAT-INFO OF WBY21W NOT = " "  
      move 5 to i  
      GO TO EZECONDLBL-00004  
    END-IF  
    GO TO EZECONDLBL-00005  
    CONTINUE.  
  EZECONDLBL-00004.
```

```
PERFORM WBY21P11-FORMAT-M1.  
CONTINUE.  
EZECONDLBL-00005.  
    PERFORM WBY21P12-CONV-M1.  
    PERFORM WBY21P13-RESP-M1.  
CONTINUE.  
EZECONDLBL-00003.  
    PERFORM WBY21P19-TERM.  
CONTINUE.  
EZE-WBY21P00-MAIN-X.  
GOBACK.  
  
WBY21P10-INIT.  
    move 1 to i.  
WBY21P11-FORMAT-M1.  
WBY21P12-CONV-M1.  
WBY21P13-RESP-M1.  
    WBY21P19-TERM.
```

**After:**

```
PROCEDURE DIVISION.  
WBY21P00-MAIN SECTION.  
    move 1 to i  
    move 2 to i  
    IF EZEAPP = " "  
        move 3 to i  
        move 4 to i  
        IF M1-FMAT-INFO OF WBY21W NOT = " "  
            move 5 to i  
        END-IF  
    END-IF.  
END-IF.
```

**Note:** Refactoring is not available for cases in which a GO TO or PERFORM occurs at the end of an IF or EVALUATE branch.

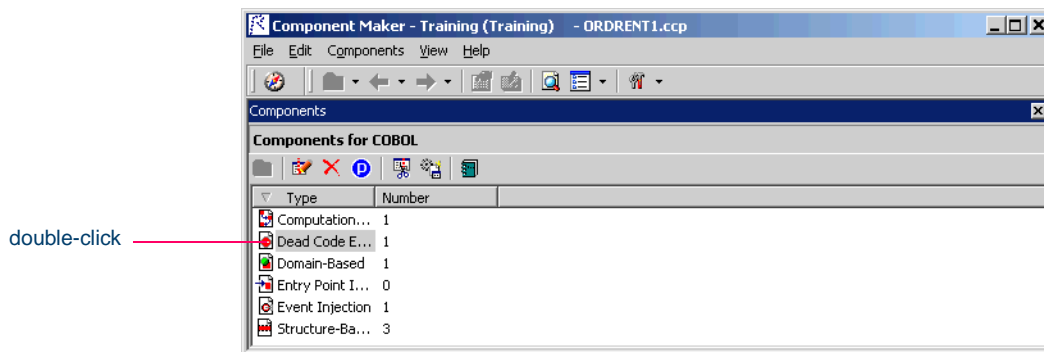
## Extracting Optimized Cobol Components

This section describes how to perform Dead Code Elimination and refactoring for Cobol programs.

### To extract an optimized Cobol component:

- 1 Start Component Maker as described in [“Starting Component Maker” on page 1-6](#). The Component Maker window opens. In the Components pane (Figure 7-1), double-click Dead Code Elimination.

Figure 7-1 *Components Pane*



- 2 The view shown in Figure 7-2 opens. This view shows the DCE-based logical components created for the programs in the current project.


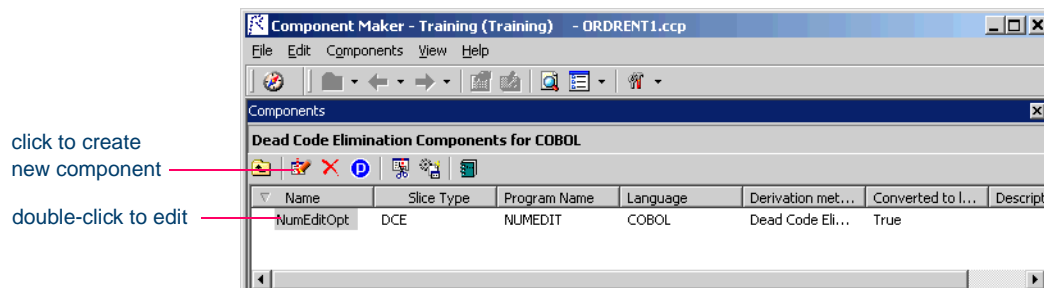
**Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.

Figure 7-2 *Components Pane — Dead Code Elimination View*




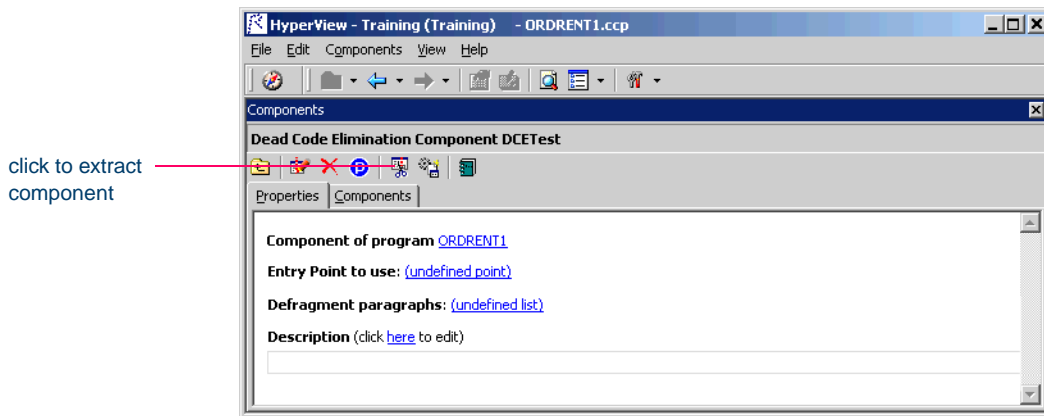
- 3A To create a new logical component, select the program you want to analyze for dead code in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 7-3 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A on page 7-7](#).

Figure 7-3 *Components Pane — Properties Tab (Cobol)*



- 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. For background, see [Chapter 8, “Performing Entry Point Isolation.”](#) To unset an entry point, click it and choose **Unset** in the pop-up menu.
- 6 In the **Defragment paragraphs** field, click the link for the current selection and choose the list of paragraphs to defragment in the pop-up menu. For background, see [“Understanding Refactoring \(Future Release\)” on page 7-3](#).

**Note:** To create a new list of paragraphs to defragment, choose **New** in the pop-up menu. The List Browser opens, where you can

**7-8** Eliminating Dead Code  
*Extracting Optimized Cobol Components*

create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.


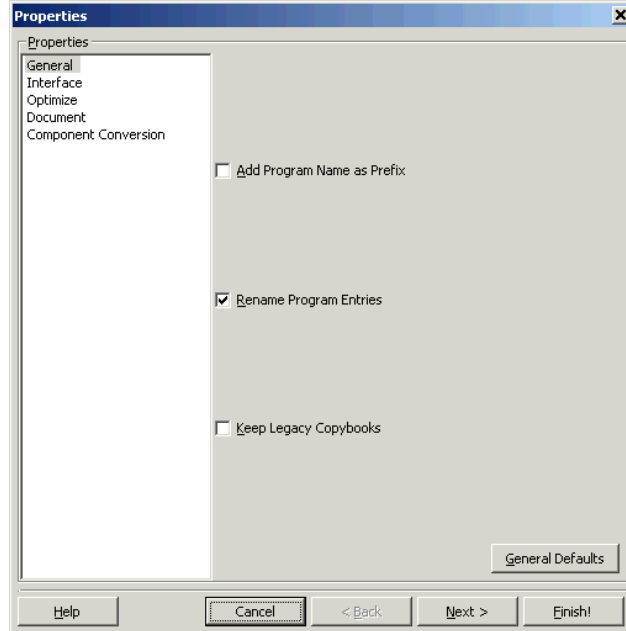
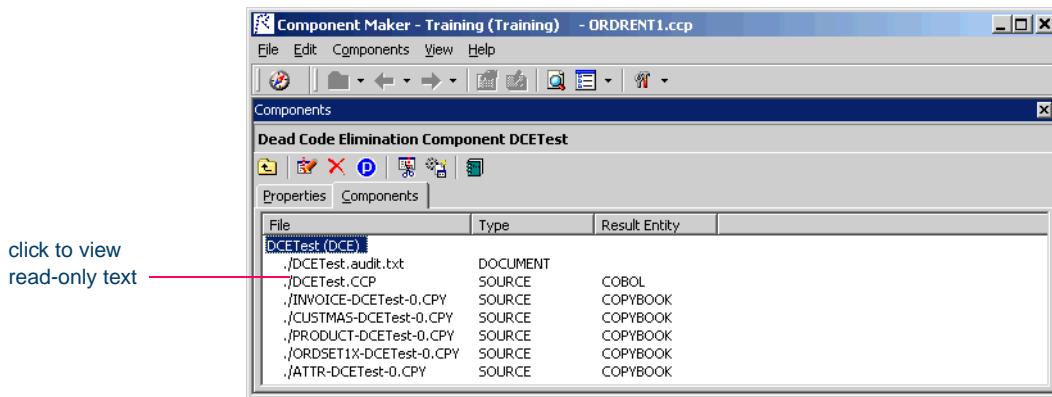
- 7 In the **Description** field, click the **here** link to open a text editor where you can enter a description of the component. The description appears in the box below the **Description** field in the Properties tab and in the Description property for the logical component repository object.
- 8 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**.
- 9 The dialog shown in Figure 7-4 opens. This dialog displays a series of panes that let you set extraction options for Dead Code Elimination. For usage information, see [“Setting Cobol Extraction Options” on page 2-2](#). When you are satisfied with your choices, click **Finish**.

Figure 7-4 *Components Pane — Extraction Options Dialog (Cobol)*



- 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 Figure 7-5 opens. This view shows a list of the component source files that were generated for the logical component and an audit report if you requested one (see [“Generating Audit Reports \(Cobol Only\)”](#) on page 1-13). Click an item in the list to view the read-only text for the item.

Figure 7-5 *Components Pane — Components Tab*



## Extracting Optimized Natural Components


This section describes how to perform Dead Code Elimination for Natural programs.

### **To extract an optimized Natural component:**

- 1 Start Component Maker as described in [“Starting Component Maker”](#) on page 1-6. The Component Maker window opens. In the Components pane (Figure 7-1 on page 7-6), double-click Dead Code Elimination.

**7-10** Eliminating Dead Code  
*Extracting Optimized Natural Components*

- 2 The view shown in Figure 7-2 on page 7-6 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.


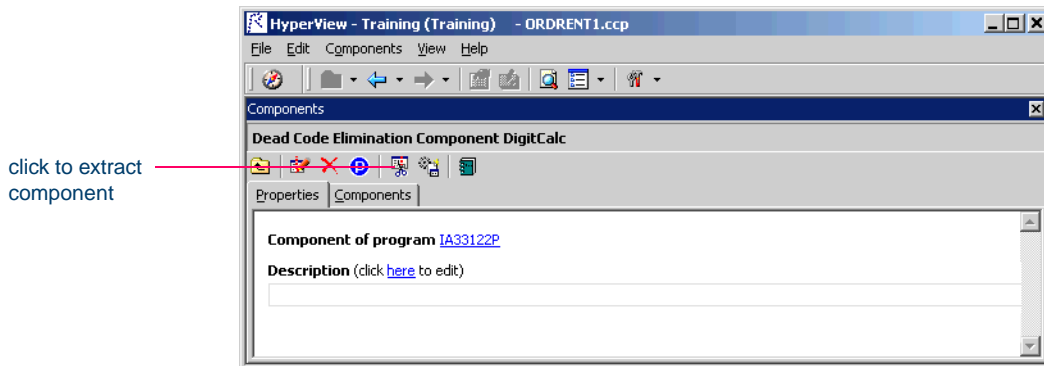
- 3A To create a new logical component, select the program you want to analyze for dead code in the Clipper pane and click the  button. A dialog opens where you can enter the name of the new component in the text field. Click **OK**.
- 3B To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4 The view shown in Figure 7-6 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A](#).

Figure 7-6 *Components Pane — Properties Tab (Natural)*



- 5 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.




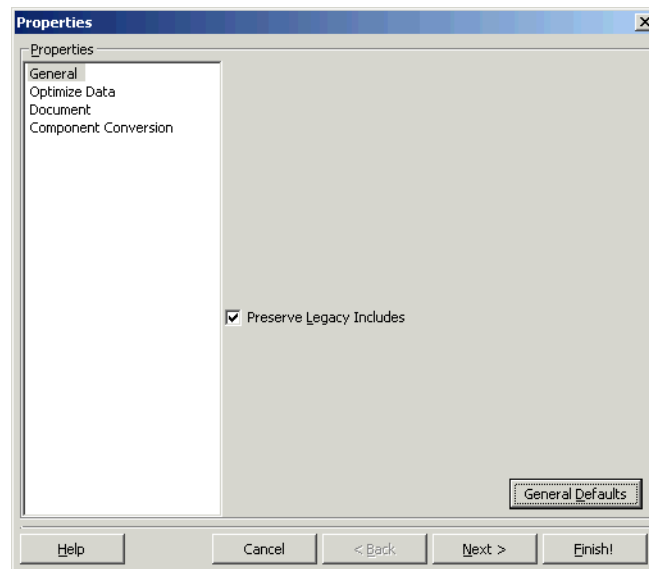
- 6 Click the  button on the tool bar to start extracting the logical component. You are prompted to confirm that you want to continue. Click **OK**.
- 7 The dialog shown in Figure 7-7 opens. This dialog displays a series of panes that let you set extraction options for Dead Code Elimination. For usage information, see [“Setting Natural Extraction Options” on page 2-15](#). When you are satisfied with your choices, click **Finish**.

Figure 7-7 *Components Pane — Extraction Options Dialog (Natural)*



- 8 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**.
- 9 Assuming the extraction executed without errors, the view shown in Figure 7-5 on page 7-9 opens. This view shows a list of the component source files that were generated for the logical component. Click an item in the list to view the read-only text for the item.

**7-12** Eliminating Dead Code  
*What's Next?*

## ***What's Next?***

That's all you need to know to perform Dead Code Elimination. Now let's look at how you use Component Maker to perform Entry Point Isolation. That's the subject of the next chapter.

## *Performing Entry Point Isolation*



**E**ntry Point Isolation lets you build a component based on one of multiple entry points in a legacy program — an inner entry point in a Cobol program, for example, rather than the start of the Procedure Division. Component Maker extracts only the functionality and data definitions required for invocation from the selected point.

Entry Point Isolation is built into the main methods as an optional optimization tool. It's offered separately in case you want to use it on a stand-alone basis.

**Note:** Entry Point Isolation is not available for PL/I and Natural Programs.

### *Extracting a Cobol Component with Entry Point Isolation*

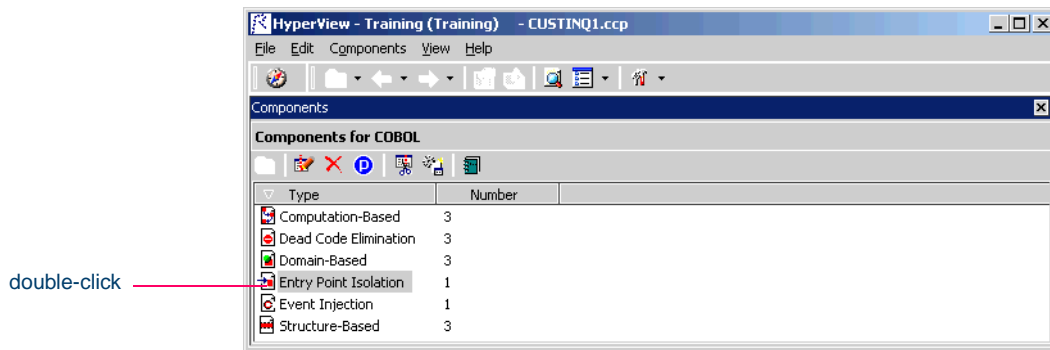
This section describes how to perform Entry Point Isolation for Cobol programs.

8-2 Performing Entry Point Isolation  
*Extracting a Cobol Component with Entry Point Isolation*

**To extract a Cobol component with Entry Point Isolation:**

- 1 Start Component Maker as described in “Starting Component Maker” on page 1-6. The Component Maker window opens. In the Components pane (Figure 8-1), double-click Entry Point Isolation.

Figure 8-1 Components Pane



- 2 The view shown in Figure 8-2 opens. This view shows the entry point isolation-based logical components created for the programs in the current project.


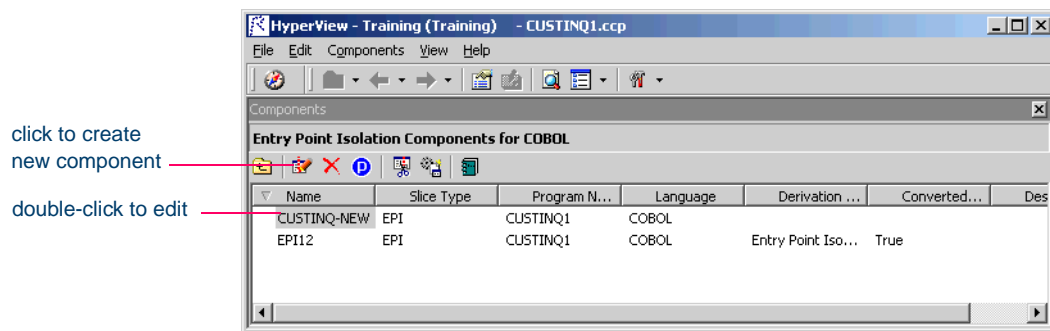

**Tip:** Click the  button on the tool bar to restrict the display to logical components created for the selected program.

Figure 8-2 Components Pane — Entry Point Isolation View

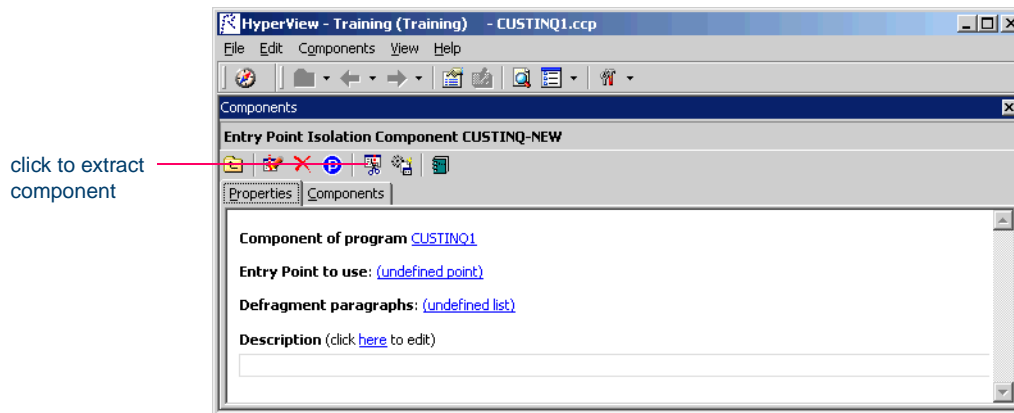


- 3A To create a new logical component, select the program you want to slice in the Clipper pane and click the  button. A dialog opens

where you can enter the name of the new component in the text field. Click **OK**.

- 3B** To edit an existing logical component — to extract it with different properties or options — double-click it.
- 4** The view shown in Figure 8-3 opens. This view shows the properties you can set for the logical component. The **Component of program** field contains the name of the program you selected in [step 3A on page 8-2](#).

Figure 8-3 *Components Pane — Properties Tab (Cobol)*



- 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.
- 6** In the **Defragment paragraphs** field, click the link for the current selection and choose the list of paragraphs to defragment in the pop-up menu. For background, see [“Understanding Refactoring \(Future Release\)” on page 7-3](#).

**8-4** Performing Entry Point Isolation  
*Extracting a Cobol Component with Entry Point Isolation*


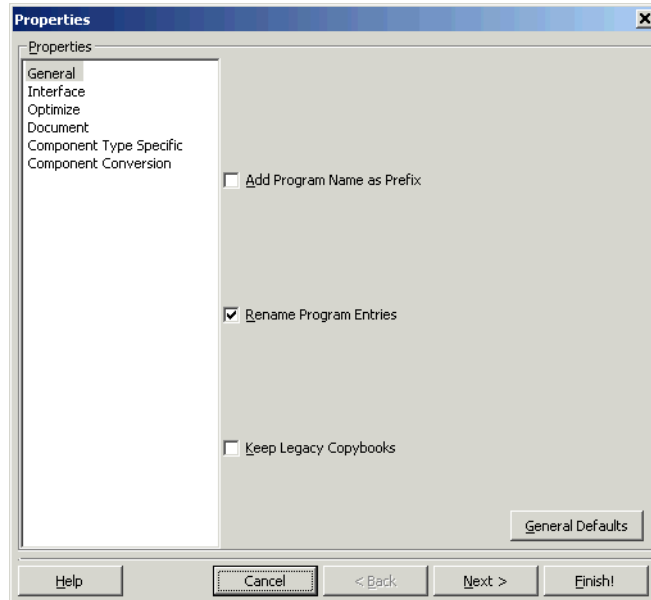
- Note:** To create a new list of paragraphs to defragment, choose **New** in the pop-up menu. The List Browser opens, where you can create the list. Choose **Show** to display the current list in the List Browser. Choose **None** to unset the list. For List Browser usage, see *Analyzing Programs* in the ATW document set.
- 7 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.
  - 8 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**.
  - 9 The dialog shown in Figure 8-4 opens. This dialog displays a series of panes that let you set extraction options for Structure-Based Componentization. For usage information, see [“Setting Cobol Extraction Options” on page 2-2](#). When you are satisfied with your choices, click **Finish**.

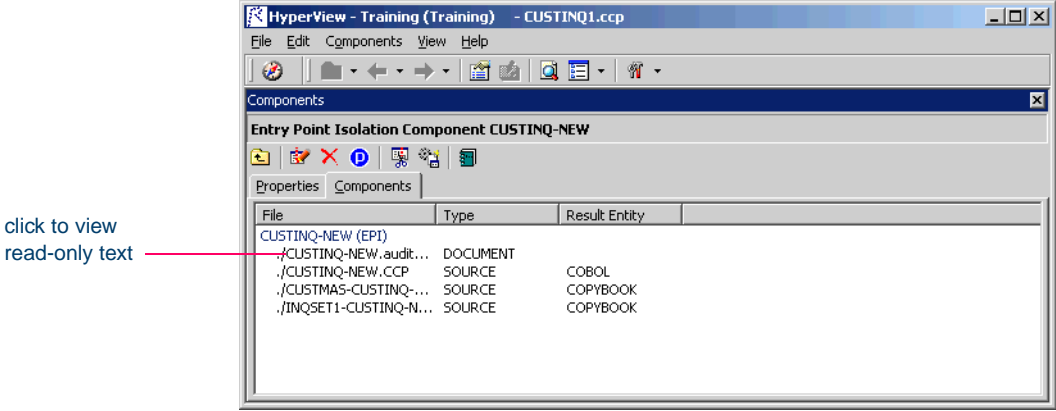
Figure 8-4 *Components Pane — Extraction Options Dialog (Cobol)*



- 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 Figure 8-5 opens. This view shows a list of the component source files that were generated for the logical component and an audit report if you requested one (see [“Generating Audit Reports \(Cobol Only\)” on page 1-13](#)). Click an item in the list to view the read-only text for the item.

**8-6** Performing Entry Point Isolation  
*What's Next?*

Figure 8-5 *Components Pane — Components Tab*



**What's Next?**

That completes your tour of Component Maker! Now you're ready to begin using the Transformation Workbench to re-architect legacy applications.



## 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; for domain-based extraction when the specialization variable is set to multiple values; 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 ATW document set.

### **Use Special IMS Calling Conventions**

Select **Use Special IMS Calling Conventions** in the Verification tab of the Project Options window 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 Termination**

Select **Override CICS Program Terminations** in the Verification tab of the Project Options window 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 Verification tab of the Project Options window 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 Verification tab of the Project Options window if you are working on a project con-

taining 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 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 Verification tab of the Project Options window 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 (Cobol and Natural Only)***

The **Relaxed Parsing** option in the Verification tab of the Workspace Options window 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: “ATW assumption:”.
- For Domain-Based Componentization:
  - Data items with errors in declarations are treated as data items with unknown values.
  - Statements with errors are treated as statements that do not change values.
  - Whenever a calculation error occurs, the comment “Calculation has not been completed successfully by ATW” is generated in the component before the erroneous operator, along with an error message.
  - Component Maker ignores user values for duplicated identifiers (which may have an association with a wrong DECL); structures with fields marked with errors; and undeclared identifiers are ignored. A list of ignored values appears at the top of the component.
  - Users cannot specify values for VARs with attribute errors (duplicated identifiers); VARs without DECLs (undeclared identifiers); and DECLs with attribute errors.

### ***PERFORM Behavior for MicroFocus Cobol***

For MicroFocus Cobol applications, use the PERFORM Behavior option in the Verification tab of the Workspace Options window to specify the type of PERFORM behavior the application was compiled with. You can select:

- **MicroFocus** if the application was compiled by the Microfocus compiler with the default Microfocus-style option set. This option requires that applications that use nested PERFORM statements complete the innermost nested PERFORM before returning control to the previous PERFORM.
- **IBM family** if applications that use nested PERFORM statements do not need to complete the innermost nested PERFORM before returning control to the previous PERFORM.

For IBM-family 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 this option to IBM-family PERFORM behavior where appropriate allows the ATW 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***

Place a check mark next to **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]
```

**Comment** 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]
```

**Comment** 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]
```

**Comment** When the option is turned on, 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 of input and output parameters are:

- **Input**

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



DURE DIVISION USING is classified as output parameter if it is modified during the component execution.

A system variable (field of DFHEIB/DFHEIBLK structures) is classified as output parameter if the **Create CICS Program** option is turned off and the variable is modified during the 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, a 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-*[Number]* 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 a code to copy all output parameters from working-storage variables to the corresponding fields of BRE-OUTPUT-STRUCTURE.

Variables of any levels (rather than only 01-level structures together with all their fields) can act as parameters. This enables to exclude unnecessary parameters and make the resulting programs more compact and clear.

For each operator, for which a parameter list is generated, the following transformations are applied to that 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 re-named 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.

***Example — Single-Value Case***

```
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.

***Example — Multiple-Value case***

```
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.

**Note:** The statement interpretation capability is available only when you define the specialization variable “positively” — as equaling 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 (COBOL Only)***

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 arithmetic exception conditions take place:

- The value of an arithmetic operation result is larger than the result-ant-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.

Specializer 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.

# Glossary

## Activity Log

The Activity Log is a chronological record of your activities in the current [Asset Transformation Workbench \(ATW\)](#) session.

## ADABAS

ADABAS is a Software AG relational [DBMS](#) for large, mission-critical applications.

## Animator

Animator lets you step through the code displayed in a [HyperView](#) pane. You can choose program branches yourself, or have the animator choose them randomly.

## API

API stands for application programming interface, a set of routines, protocols, and tools for building software applications.

## applet

See [Java applet](#).

## Application Analyzer

Application Analyzer is a set of non-invasive interactive tools used to analyze and document legacy systems.

### **Application Architect**

Application Architect uses advanced algorithms to partition code into new [components](#) and perform [Dead Code Elimination](#).

### **Application Namespace tool**

The Application Namespace tool creates a conveniently organized dictionary that helps you navigate through your system's terminology and modify it as necessary.

### **Application Partitioner**

Application Partitioner identifies legacy subsystems and partitions them into self-contained projects based on an analysis of [repository](#) contents.

### **Application Profiler**

Application Profiler, consisting of [WebGen](#) and [Profiler](#), generates a set of HTML views of a legacy application based on the [object model](#) created in a previous analysis.

### **AS/400**

The AS/400 is a midrange server designed for small businesses and departments in large enterprises.

### **Asset Transformation Workbench (ATW)**

Asset Transformation Workbench (ATW) is a suite of PC-based software products for analyzing, re-architecting, and transforming legacy applications.

### **Batch Application Viewer**

Batch Application Viewer performs low-level analysis of batch processes.

### **batch refresh**

The batch refresh feature lets you register and verify source files in batch mode. Other utilities packaged with the feature let you analyze application [complexity](#), run [WebGen](#), and, if you are licensed to use the Application Architect product, perform [Dead Code Elimination](#).

### **Bird's Eye pane**

The Bird's Eye pane works with the [HyperView](#) Source pane to let you quickly identify the location of a code construct relative to the entire program.

**BMS**

BMS stands for Basic Mapping Support, an interface between application formats and [CICS](#) that formats input and output display data.

**BSTR**

BSTR is a Microsoft format for transferring binary strings.

**business rule**

A business rule is a named container that identifies and documents code segments according to their business function. Business rules encapsulate an application's business logic, making the application easier to understand, document, maintain, and test.

**Business Rule Manager**

Business Rule Manager lets you generate [business rules](#) from code segments extracted manually from source or autodetected.

**Callie pane**

The [HyperView](#) Callie pane displays a diagram that shows the flow of control between paragraphs or procedures in a program.

**CDML**

CDML stands for Cobol Data Manipulation Language, an extension of the [Cobol](#) programming language that enables applications programmers to code special instructions to manipulate data in a [DMS](#) database and to compile those instructions for execution.

**Change Analyzer**

Change Analyzer identifies the class of data items used to perform a business function in a legacy application. Among other uses, it lets you answer the kinds of "What if?" questions posed in the recent past by the industry-wide changes for Y2K, Zip+4, and the Euro dollar: "What if I change the type of this variable, or the length of this field — what *other* fields will I also have to change?"

**CICS**

CICS stands for Customer Information Control System, a program that allows concurrent processing of [transactions](#) from multiple terminals.

### **Clipper**

The [HyperView](#) Clipper tool lets you create lists of candidates for [business rule](#) extraction, [event injection](#), and other tasks. Each list captures the results of a different stage of your analysis and serves as input for subsequent tasks.

### **Cobol**

Cobol stands for Common Business-Oriented Language, a high-level programming language used for business applications.

### **COM**

COM stands for Component Object Model, a software architecture developed by Microsoft to build [component](#)-based applications. COM objects are discrete components, each with a unique identity, which expose interfaces that allow applications and other components to access their features.

### **complexity**

A [project's](#) complexity is an estimate of how difficult it is to maintain, analyze, transform, and so forth.

### **component**

A component is a self-contained program that can be reused with other programs in modular fashion.

### **Component Maker**

The [HyperView](#) Component Maker tool lets you “slice out” [components](#) from legacy applications — not only component executables but associated [Cobol](#) copybooks, [PL/I](#) includes, and [Natural](#) data areas as well.

### **computation-based component extraction**

Computation-based [component](#) extraction lets you build a component that contains all the code necessary to calculate the value of a variable at a particular point in a program — the value of a DayOfTheWeek variable, for example, where it is used to populate a report attribute or screen.

### **Configuration Manager**

Configuration Manager is a tool used to enable [Asset Transformation Workbench \(ATW\)](#) products and configure your workbench for the



tools, programming languages, legacy dialects, and character sets in use at your site.

**construct**

A construct is an item in the [parse tree](#) for a source file — a section, statement, condition, variable, or the like. A variable, for example, can be related in the parse tree to any of three other constructs — a declaration, a dataport, or a condition. You view the parse tree for a source file in the HyperView [Context pane](#).

**Context pane**

The [HyperView](#) Context pane displays the [parse tree](#) for the selected source file. The parse tree displays source code constructs — sections, paragraphs, statements, conditions, variables and so forth — in hierarchical form, making it easy to locate code constructs quickly.

**copybook**

A copybook is a common piece of source code to be copied into many [Cobol](#) source programs. Copybooks are functionally equivalent to C and C++ include files.

**CORBA**

CORBA stands for Common Object Request Broker Architecture, an architecture that enables distributed objects to communicate with one another regardless of the programming language they were written in or the operating system they are running on.

**CSD file**

CSD stands for [CICS](#) System Definition. A CSD file is a [VSAM](#) data set containing a resource definition record for every resource defined to [CICS](#).

**database schema**

A database schema is the structure of a database system, described in a formal language supported by the [DBMS](#). In a relational database, the schema defines the tables, the fields in each table, and the relationships between fields and tables.

**dataport**

A dataport is an input/output statement or a call to or from another program.

### **DB/2**

DB/2 stands for Database 2, an IBM system for managing relational databases.

### **DBCS**

DBCS stands for double-byte character string, a character set that uses two-byte (16-bit) characters rather than one-byte (8-bit) characters.

### **DBMS**

DBMS stands for database management system, a collection of programs that enable you to store, modify, and extract information from a database.

### **DDL**

DDL stands for Data Description Language (DDL), a language that describes the structure of data in a database.

### **Dead Code Elimination**

Dead code elimination is a type of [component](#) extraction that removes unused (“dead”) code from a legacy application.

### **decision resolution**

Decision resolution lets you identify and resolve dynamic calls and other relationships that the [parser](#) cannot resolve from static sources.

### **Diagrammer**

Diagrammer lets you view the [relationships](#) between the objects in a [project](#) interactively — programs, files, [DDL](#), [Java](#), screen maps, and more. These relationships describe the ways in which application objects interact. Compare [Quick Diagrammer](#).

### **DMS**

DMS stands for Data Management System, a Unisys database management software product that conforms to the CODASYL (network) data model and enables data definition, manipulation, and maintenance in mass storage database files.

### **domain-based component extraction**

Domain-based component extraction “specializes” a program based on the values of one or more variables. The specialized program is typically intended for reuse “in place” — in the original application but under new external circumstances.

**DPS**

DPS stands for Display Processing System, a Unisys product that enables users to define forms on a terminal.

**ECL**

ECL stands for Executive Control Language, the operating system language for Unisys OS 2200 systems.

**effort**

Effort is an estimate of the time it will take to complete a task related to a [project](#), based on weighted values for selected [complexity](#) metrics.

**EJB**

EJB stands for Enterprise JavaBeans, a [Java API](#) developed by Sun Microsystems that defines a [component](#) architecture for multi-tier client/server systems.

**EMF**

EMF stands for Enhanced MetaFile, a Windows format for graphic images.

**entity**

An entity is an object in the [repository](#) model for a legacy application. The relationships between entities describe the ways in which the elements of the application interact.

**entry point isolation**

Entry point isolation extracts a [component](#) that contains only the functionality and data definitions required for invocation from the selected entry point.

**event injection**

Event injection is a type of [component](#) extraction that adapts a legacy program to asynchronous, event-based programming models.

**Execution Path pane**

The [HyperView](#) Execution Path pane displays a hierarchical view and diagram of the conditions that determine the flow of control in a program.

### **external subroutine extraction**

External subroutine extraction is a type of [structure-based component extraction](#) that replaces a single internal subroutine in a [Natural](#) program with an external subroutine.

### **FCT**

FCT stands for File Control Table (FCT), a [CICS](#) table that contains processing requirements for output data streams received via a remote job entry session from a host system. Compare [PCT](#).

### **Flowchart pane**

The [HyperView](#) Flowchart pane displays a diagram of the flow of control between statements in a paragraph or procedure.

### **Global Data Flow tool**

The [HyperView](#) Global Data Flow tool performs low-level analysis of program data flows.

### **HTML**

HTML stands for HyperText Markup Language, the authoring language used to create documents on the World Wide Web.

### **HyperView**

HyperView is a set of program analysis tools that let you analyze legacy programs interactively, by examining synchronized, complementary views of the same information — source, context, impacts, and so forth.

### **IDL**

IDL stands for Interface Definition Language (IDL), a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language.

### **IDMS**

IDMS stands for Integrated Database Management System, a Computer Associates database management system for the IBM mainframe and compatible environments.

### **Impact pane**

The [HyperView](#) Impact pane displays a hierarchical view and diagram of the *impact trace* for a program variable. An impact trace describes

how data items interact with each other in a program — exchange values, use each other in computations, and so forth.

**Impact Report pane**

The [HyperView](#) Impact Report pane shows the flow of data from a startup item to every data item that would be affected by its modification. The report is organized in hierarchical form according to the depth of the affected item.

**IMS**

IMS stands for Information Management System, an IBM program product that provides transaction management and database management functions for large commercial application systems.

**Java**

Java is a high-level [object-oriented programming](#) language developed by Sun Microsystems.

**Java applet**

A [Java](#) applet is a program that can be sent with a Web page. Java applets perform interactive animations, immediate calculations, and other simple tasks without having to send a user request back to the server.

**JavaBeans**

JavaBeans is a specification developed by Sun Microsystems that defines how [Java](#) objects interact. An object that conforms to this specification is called a JavaBean.

**JCL**

JCL stands for Job Control Language, a language for identifying a [job](#) to OS/390 and for describing the job's requirements.

**JDBC**

JDBC stands for Java Database Connectivity, a standard for accessing diverse database systems using the [Java](#) programming language.

**job**

A job is the unit of work that a computer operator or a program called a *job scheduler* gives to the operating system. In IBM mainframe operating systems, a job is described with job control language ([JCL](#)).

### **job dependencies**

[Batch Application Viewer](#) treats [jobs](#) as dependent if one writes to a dataset and the other reads from the same dataset. Occasionally, you may want to define dependencies between jobs based on other criteria — administrative needs such as scheduling, for example.

### **logical component**

A logical component is an abstract [repository](#) object that gives you access to the source files that comprise a [component](#).

### **MFS**

MFS stands for Message Format Service, a method of processing [IMS](#) input and output messages.

### **Missing Copybooks Resolution tool**

The Missing Copybooks Resolution tool resolves undefined variables in missing [copybooks](#) for [Cobol](#) programs verified with the [relaxed parsing](#) option.

### **Model Reference pane**

The [HyperView](#) Model Reference pane displays the [parse tree](#) meta-model in text and diagram form.

### **name-based partitioning**

Name-based partitioning is a partitioning algorithm that assigns source files to projects based on text matching of source file names with specified patterns.

### **Natural**

Natural is a programming language developed and marketed by Software AG for the enterprise environment.

### **object model**

An object model is a representation of an application and its encapsulated data.

### **object-oriented programming**

Object-oriented programming organizes programs in terms of objects rather than actions, and data rather than logic.

**ODBC**

ODBC stands for Open Database Connectivity, a standard for accessing diverse database systems.

**orphan**

An orphan is an object that does not exist in the reference tree for any startup object. Orphans can be removed from a system without altering its behavior.

**Orphan Analysis tool**

The Orphan Analysis tool lets you analyze and resolve [orphans](#).

**parser**

The [Asset Transformation Workbench \(ATW\)](#) parser defines the [object model](#) and [parse tree](#) for a legacy application.

**parse tree**

A parse tree defines the relationships among the constructs that comprise a source file — its sections, paragraphs, statements, conditions, variables, and so forth.

**PCT**

PCT stands for Program Control Table, a [CICS](#) table that defines the transactions that the CICS system can process. Compare [FCT](#).

**PL/I**

PL/I stands for Programming Language One, a third-generation programming language developed in the early 1960s as an alternative to assembler language, [Cobol](#), and FORTRAN.

**PL/I Call Diagrammer**

PL/I Call Diagrammer performs low-level analysis of [PL/I](#) programs. Use it to examine call flows for internal procedures that the [Diagrammer](#) is unable to model.

**profile**

Profiles are HTML views into a [repository](#) that show all of the analysis you have done on an application. Profiles are convenient ways to share information about legacy applications across your organization.

### **Profiler**

Profiler is a Web server-based tool that offers company-wide access to [profiles](#) of any repository in your organization. It gives managers, business analysts, testers, and customer support personnel convenient, browser-based access to analyzed legacy code.

### **project**

A project is a logical subdivision of a [workspace](#). You might have a project for the batch portion of your application and another project for the online portion, for example. You can also use a project to collect items for discrete tasks — all the source files affected by a change request, for example.

### **QSAM**

QSAM stands for Queued Sequential Access Method, a type of processing that uses a queue of data records—either input records awaiting processing or output records that have been processed and are ready for transfer to storage or an output device.

### **Quick Diagrammer**

The Quick Diagrammer tool lets you view relationships for selected objects only, rather than an entire project. Compare [Diagrammer](#).

### **refactoring**

Refactoring translates a program into a [component](#) with the same functionality and control flow, but a simpler syntax structure.

### **reference reports**

[Asset Transformation Workbench \(ATW\)](#) offers three related reports that you can use to identify missing or unneeded program elements in application source: an unresolved report, an unREFERRED report, and a cross-reference report.

### **relationship**

The relationships between entities in the [repository](#) model for a legacy application describe the ways in which the elements of the application interact.

### **relationship-based partitioning**

Relationship-based partitioning is a partitioning algorithm that assigns source files to projects based on the extent to which the source



files are related. If two source files reference the same [copybook](#), for example, they can be regarded as “tightly related,” at least as compared with source files that do not reference the same copybook.

**relationship weight**

A relationship weight determines the importance of that [relationship](#) in calculating the connection between source files in [relationship-based partitioning](#).

**relaxed parsing**

Relaxed parsing 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.

**repository**

A repository is a database of program objects that comprise the model for a [workspace](#).

**Repository Browser**

The [Asset Transformation Workbench \(ATW\)](#) Repository Browser displays the contents of the current [workspace](#).

**Resource Retriever**

The Resource Retriever tool lets you identify and restore missing [CICS](#) file connectors and transactions in file control tables ([FCT](#)) and program control tables ([PCT](#)) for [Cobol](#) and [PL/I](#) programs.

**Rules pane**

The [HyperView](#) Rules pane lets you create [business rules](#) from code segments extracted manually from source or autodetected. You can also create business rules from candidates listed in [Clipper](#).

**schema**

See [database schema](#).

**scope**

The scope of a diagram determines the objects and [relationships](#) it displays. See [Diagrammer](#).

**seed field**

A seed field is the object of a [Change Analyzer](#) search for the class of data items that need to be changed.

### Source pane

The [HyperView](#) Source pane displays view-only source code for the selected file and included files.

### SQL

SQL stands for Structured Query Language, a standard language for relational database operations

### structure-based component extraction

Structure-based component extraction is a type of [component](#) extraction that builds a component from a range of inline code — [Cobol](#) paragraphs, for example.

### synonym

A synonym is a data field whose value is related to the value of the matched [seed field](#) — a field whose value is assigned by a MOVE or REDEFINE statement, for example.

### system program

A system program is a generic program — a mainframe sort utility, for example — provided by the underlying system and used in unmodified form in the legacy application.

### TIP

TIP stands for Transaction Processing, the Unisys real-time system for processing transactions under Exec control.

### token

In the [Application Namespace tool](#), a token is an element in a program identifier delimited by a hyphen (-) or underscore (\_). In the identifier WS01-CUST-FIELD, for example, there are three tokens: WS01, CUST, and FIELD.

### transaction

A transaction is a sequence of information exchange and related work (such as database updating) that is treated as a unit for the purposes of satisfying a request and for ensuring database integrity.

### Unknown Statements Resolution tool

The Unknown Statements Resolution tool resolves incorrect or unsupported statements in [Cobol](#) programs verified with the [relaxed parsing](#) option.

**User Interface tool**

The User Interface tool lets you analyze the interaction between legacy screens and program logic, and generate an [HTML](#) or [Java](#) GUI based on the interaction.

**VALTAB**

VALTAB stands for Validation Table, which contains the information the system needs to locate, load, and execute transaction programs. See also [TIP](#).

**VSAM**

VSAM stands for Virtual Storage Access Method, an IBM program that controls communication and the flow of data in a Systems Network Architecture network.

**WebGen**

WebGen generates HTML views of the repositories on your workstation. You can publish the views to [Profiler](#), where they can be accessed by any member of your organization with a browser.

**workspace**

A workspace is a named container for an application or a portion of an application. Workspaces can be divided into [projects](#).

**XML**

XML stands for Extensible Markup Language, a specification for creating common information formats.



---

# **Bibliography**

- *IBM Asset Transformation Workbench v1.1 Getting Started (SC31-6877-00)*
- *IBM Asset Transformation Workbench v1.1 Preparing Projects (SC31-6879-00)*
- *IBM Asset Transformation Workbench v1.1 Analyzing Projects (SC31-6880-00)*
- *IBM Asset Transformation Workbench v1.1 Analyzing Programs (SC31-6878-00)*
- *IBM Asset Transformation Workbench v1.1 Profiling Projects (SC31-6881-00)*
- *IBM Asset Transformation Workbench v1.1 Creating Components (SC31-6876-00)*
- *IBM Asset Transformation Workbench v1.1 Parser Reference (SC31-6882-00)*
- *IBM Asset Transformation Workbench v1.1 Architecture Reference (SC31-6898-00)*



---

# Notices

This information was developed for products and services offered in the U.S.A. IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing

IBM Corporation

North Castle Drive

Armonk, NY 10504-1785

U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation

Licensing

2-31 Roppongi 3-chome, Minato-ku

Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION •AS IS• WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
P.O. Box 12195, Dept. TL3B/B503/B313  
3039 Cornwallis Rd.  
Research Triangle Park, NC 27709-2195  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.



---

## Trademarks

The following terms are trademarks of the IBM Corporation or its subsidiaries in the United States or other countries or both:

*Table 1. Trademarks*

IBM	MVS
AS400	CICS
IMS	DB/2
Database 2	OS/390
S/390	z/OS

The following terms are trademarks of other companies:

Java and JavaScript are registered trademarks and Sun Solaris and Solaris are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.



# Index

## A

arithmetic exception handling A-12  
audit reports 1-13, 2-7

## B

Batch Dead Code Elimination 7-1  
blocking 2-4, 4-2

## C

CICS components 1-15, 2-3  
Cobol extraction options 2-2  
commenting out unused code 2-8, 2-17,  
7-1  
complement 3-1, 3-2  
Component Maker  
    general usage 1-10  
    options 2-1  
    outputs 1-5

overview 1-1, 1-2  
sample usage 1-7  
starting 1-6

## Computation-Based Componentization

blocking 2-4, 4-2  
Cobol 4-3  
Natural 4-7  
overview 4-1  
parameterized slices 3-3  
required verification options 3-5, 4-1  
type-specific options 2-9  
variable- versus statement-based 2-9,  
4-2

conversion options 2-12, 2-15, 2-17  
converting components 1-5, 2-12  
copybooks, preserving versus modifying  
    2-3, 2-7, A-6  
coverage reports 1-14

## D

- data areas 2-17
- Dead Code Elimination
  - batch 7-1
  - Cobol 7-6
  - generating statistics 7-2
  - Natural 7-9
  - overview 7-1
  - type-specific options 2-4, 2-14, 2-15, 2-17
  - usage example 7-2
- defragmenting programs 7-3
- document extraction options 2-7
- Domain-Based Componentization
  - advanced mode 5-5
  - Cobol 5-7
  - lite mode 5-6
  - overview 5-1
  - PL/I 5-14
  - simplified mode 5-2
  - type-specific options 2-10
- dynamic call 2-9

## E

- Entry Point Isolation
  - overview 8-1
  - performing 8-2
  - renaming entry points 2-3
- Event Injection
  - overview 6-1, 8-1
  - performing 6-4
  - sample usage 6-2
  - type-specific options 2-11
  - use with Clipper 6-1
- exporting components 1-15
- External Subroutine Componentization
  - 3-17

## F

- FILLERs 2-6

## G

- general extraction options 2-2, 2-13, 2-15

## H

- HTML extraction trace 2-10
- HyperView lists 1-12

## I

- IMS calling conventions A-1
- include files, preserving versus modifying 2-15
- interface extraction options 2-3

## L

- logical component
  - computation-based 4-1
  - domain-based 5-1
  - eliminating dead code 7-1
  - event-injected 6-1, 8-1
  - exporting 1-15
  - isolating entry points 8-1
  - overview 1-5

## M

- MAP file 2-17
- marking modified code 2-8
- MicroFocus Cobol PERFORM behavior
  - A-5
- MQ Series 2-11, 6-1

## N

- nested IFs 2-5

**O**

optimization  
  extraction options 2-4, 2-15  
  overview 7-1

## options

  Cobol 2-2  
  Natural 2-15  
  overview 2-1  
  PL/I 2-13  
  verification 2-1

**P**

parameterized slices  
  extracting 2-3, 2-14  
  overview 3-1, 3-3  
  technical details A-8  
PL/I extraction options 2-13  
program specialization 5-1

**R**

ranges 3-1, 3-8  
reason codes 1-13, 2-7  
refactoring  
  options 2-5  
  overview 7-3  
  performing 7-6  
registration 1-5  
relaxed extraction 2-9  
relaxed parsing option A-4

**S**

specializing programs 5-1  
Structure-Based Componentization  
  Cobol 3-6  
  Natural 3-14, 3-17  
  parameterized slices 3-3  
  PL/I 3-11

  ranges 3-1

  type-specific options 2-8, 2-14, 2-16

subroutine 3-17

support comments 2-7

**U**

Unisys common-storage analysis A-3  
Unisys TIP and DPS calls analysis A-2  
unreachable code, removing 2-4, 2-14  
unused data items, removing 2-6, 2-15

**V**

verification 1-5  
verification options 2-1, 3-5, 4-1, A-1

Index-4





Product Number: 5724-L54

SC31-6876-00



(1P) P/N:5724-L54

