

Rational Environment User's Guide

Copyright © 1987 by Rational

Document Control Number: 8001A-05
Rev. 1.0, November 1987 (D_9_25_1)

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

Rational and R1000 are registered trademarks and Rational Environment and Rational Subsystems are trademarks of Rational.

VT100 is a trademark of Digital Equipment Corporation.

Rational
1501 Salado Drive
Mountain View, California 94043

Contents

How to Use This Book	xi
Key Concepts: Rational Environment	xiii

Part I. Getting Started

Chapter 1. The Rational Terminal Keyboard	I-1
The Rational Terminal Keyboard Layout	I-2
Main Keyboard	I-4
Function Keys	I-4
Numeric Keypad	I-4
Cursor Keys	I-5
Modifier Keys	I-5
Item Keys	I-5
Auxiliary Keys	I-6
Executing Key Combinations	I-6
Item-Operation Key Combinations	I-6
Executing an Item-Operation Key Combination	I-6
Modified Key Combinations	I-7
Executing a Modified Key Combination	I-7
The Rational Terminal Keyboard Overlay	I-8
How the Keyboard Overlay Is Organized	I-8
Reading the Keyboard Overlay	I-8
Summary of Key Notation	I-10
Symbols	I-10
Numeric Arguments	I-10
Case of Keys	I-11
Chapter 2. Logging In and Logging Out	I-13
Logging Into the Rational Environment	I-13

The Basic Login Process	I-13
Logging Into Nondefault Sessions	I-15
Logging Into Multiple Sessions	I-15
Checking the Terminal Type	I-16
Changing Your Password	I-17
Logging Out	I-18
Logging Out with Unsaved Changes	I-18
Chapter 3. Traversing the Rational Environment	I-21
What You See When You Log In	I-21
Message Window	I-22
Major Window	I-22
Home Library	I-22
Command Window	I-23
Customized Library Display Format	I-23
Customized Login Display	I-24
Moving between and within Windows	I-24
Objects in the Environment	I-25
Files	I-25
Ada Compilation Units	I-26
Libraries (Worlds and Directories)	I-27
Environment Objects and Access Control	I-28
The Environment Library Structure	I-28
Fully Qualified Object Names	I-30
The Current Context in the Library Hierarchy	I-30
Traversing the Environment Library Structure	I-31
Traversing from a Library to an Object in It	I-32
Traversing between Ada Specifications and Bodies	I-36
Returning Home	I-38
Traversing to the Enclosing Library	I-40
Traversing the Environment: Summary	I-42
Chapter 4. Managing Windows	I-45
Windows and Images	I-45
Scrolling an Image	I-45
Window Banners	I-47
Modification Symbols in the Window Banner	I-48
How Windows Are Placed on the Screen	I-49
Windows and Frames	I-49

Moving between Windows	I-50
Default Window Placement	I-51
Controlling Window Placement by Locking Windows	I-51
Controlling Window Placement through Traversal Commands	I-52
Redisplaying Windows Using the Window Directory	I-52
Displaying the Window Directory	I-53
Redisplaying Replaced Windows	I-54
Checking the Window Directory Before Logging Out	I-55
Changing Window Size and Placement	I-56
Joining Frames	I-56
Expanding Windows	I-57
Shrinking Windows	I-58
Making Frame Sizes Equal	I-58
Removing Windows from the Screen	I-58
Rearranging Windows on the Screen	I-59
Changing the Number of Frames	I-60
Chapter 5. Executing Commands	I-63
Using Command Windows	I-64
Unrecognized Commands	I-67
Correcting Typing Errors	I-68
Canceling Command Execution	I-68
Command Windows and Attached Windows	I-68
Ada Usage in Command Windows	I-69
Entering Parameters	I-69
Using the Command Window Declare Block	I-71
Getting Prompting Assistance	I-72
Getting Formatting Assistance through the Promote Key	I-72
Getting Semantic Completion through the Complete Key	I-72
Filling In Parameter Prompts	I-73
Moving between Prompts	I-73
Completing Ambiguous Name Fragments	I-74
Completion Menu Entries	I-75
Abbreviating Commands	I-76
Clearing a Command Window	I-76
Reusing Command Windows	I-77
Executing Subsequent Commands	I-77
Reexecuting the Same Command	I-77
Modifying and Reexecuting Commands	I-78

Recalling Previous Commands	I-79
Keys and Command Windows	I-79
Environment Commands	I-80
Fully Qualified Ada and Environment Names	I-80
Visibility in Command Windows	I-81
Special Names and Parameter Placeholders	I-81
Chapter 6. Getting Help	I-83
How Do the Help Keys Work?	I-83
What Command Does This Key Execute?	I-84
Reading Help Messages	I-84
What Does This Command Do?	I-85
What Commands Pertain to This Topic?	I-86
Reading Help Menus	I-87
Getting Further Help from a Menu	I-88

Part II. Editing Text

Chapter 7. Creating and Saving Text Files	II-1
Creating Text Files	II-1
Entering Text	II-3
Saving Changes as You Edit	II-4
Versions of Files	II-4
Discarding Changes Since the Last Save	II-4
Opening Existing Files for Editing	II-5
Example 1: Opening a Displayed File	II-5
Example 2: Opening a Selected File	II-5
Write Locks	II-6
Closing Files for Editing	II-7
Chapter 8. Modifying Text	II-9
Adding Text	II-9
Operating on Designated Text Items	II-10
Text Items	II-10
Patterns in Editing Operations	II-11
Selecting Text Items	II-11
Selecting an Arbitrary Stretch of Text	II-11
Using Object Selection	II-12
Selecting within a File's Structural Hierarchy	II-12
Turning Selections Off	II-13

Summary of Selection Operations	II-13
Deleting Text	II-14
Retrieving Deleted Text	II-14
Copying and Moving Text	II-15
Copying Text	II-15
Duplicating a Line	II-15
Moving Text	II-16
Summary of Copy and Move Operations	II-16
Transposing Text	II-16
Searching and Replacing	II-17
Example: Searching for a String	II-18
Summary of Search and Replace Operations	II-19
Controlling Case and Text Format	II-20
Changing Character Case	II-20
Adjusting Text Format	II-21
Setting Word Wrap for Text	II-21
Filling Existing Lines of Text	II-22
Justifying Text	II-22
Centering Lines	II-23
Changing the Fill Column	II-23
Inserting Page Breaks	II-23

Part III. Developing Simple Ada Programs

Chapter 9. Overview of Ada Unit Development	III-1
Ada Compilation Units	III-2
Ada Unit States	III-3
Source State	III-3
Installed State	III-3
Coded State	III-4
The Environment's Compilation System	III-5
A Sample Library	III-6
Example: Creating and Executing an Ada Procedure	III-7
Chapter 10. Creating, Saving, and Promoting Ada Units	III-13
Creating Ada Units	III-13
Determining Ada Unit Names and Subclasses	III-14
Creating Subprograms	III-15
Creating Package Specifications and Bodies	III-15

Saving Work in Progress	III-17
Discarding Changes Since the Last Save	III-17
Opening Existing Units for Editing	III-18
Write Locks	III-18
Versions of Units	III-19
Closing Source Units for Editing	III-20
Promoting Units to the Installed State	III-20
Installing Units with Dependencies	III-22
Reading the Compilation Log	III-24
Overview of Operations for Changing Unit State	III-25
Changing to a Relative State	III-25
Changing to a Specific State	III-25
Changing the State of a System of Units	III-26
Chapter 11. Using Ada-Specific Editing Operations	III-27
Using the Format Key	III-27
Example: Using Format to Enter a Function	III-28
Hints for Using the Format Key	III-32
Checking for Semantic Errors	III-36
Syntactic and Semantic Error Reporting	III-37
Selecting Ada Constructs	III-39
Kinds of Selection Operations	III-39
Selecting Larger or Smaller Ada Constructs	III-40
Selecting the Next or Previous Ada Constructs	III-42
Creating Private Parts	III-43
Creating Bodies	III-45
Entering Comments	III-47
Operations for Entering Comments	III-47
Inserting Page Breaks	III-48
Chapter 12. Executing and Testing Ada Programs	III-49
Promoting Units to the Coded State	III-49
Coding Individual Units	III-49
Coding Units with Dependencies	III-50
Executing Programs	III-51
Using a Command Window	III-51
Using Selection	III-52
Operations for Job Control	III-52
Common Errors	III-52

Testing Units and Systems	III-53
Saving Interactive Test Programs	III-55
Chapter 13. Debugging Ada Programs	III-57
Starting the Debugger	III-58
The Debugger Window	III-59
Controlling Program Execution	III-60
Automatic Source Display	III-60
Stepping Through a Program	III-61
Following the Program's Flow of Control	III-63
Stepping Over Subprogram Calls	III-66
Setting Breakpoints	III-67
Breakpoint Characteristics	III-68
Executing to a Breakpoint	III-69
Displaying Variable Values	III-70
Modifying Variable Values	III-71
Redisplaying the Current Location	III-71
Reexecuting a Program	III-71
Catching Exceptions	III-73
Examining the Stack of Subprogram Calls	III-74
Displaying the Call Stack	III-74
Displaying Qualified Names in the Stack	III-76
Traversing from the Call Stack	III-76
Displaying Parameter Values for a Frame	III-77
When You Have Finished Debugging	III-77
Chapter 14. Browsing Ada Programs	III-79
Where Is This Defined?	III-80
If Definition Fails	III-80
Example 1: Viewing the Definition of a Subprogram	III-81
Selection versus Cursor Position	III-83
Some Browsing Options	III-83
Example 2: Viewing the Definition of a Variable	III-83
Where Is This Used?	III-86
Example 1: Showing Variable References	III-87
Example 2: Showing Usages in Multiple Units	III-88
Chapter 15. Modifying Installed or Coded Programs	III-91
Elements That Can Be Changed Incrementally	III-92
If Dependencies Exist	III-92

Units and States	III-93
Using Incremental Operations	III-93
Incrementally Modifying an Element	III-94
Selecting One or More Elements	III-96
Using the Window Provided by an Incremental Operation	III-97
Incrementally Deleting an Element	III-97
Incrementally Adding an Element	III-99
Adding a New Declaration	III-99
Adding the Corresponding Body	III-100
Determining the Kind of Element That Is Added	III-103
Some Common Problems	III-105
Removing an Unwanted Prompt	III-105
Forgetting to Demote a Body	III-105
Selecting a Construct That Cannot Be Edited	III-106
Attempting to Change a Declaration That Has Dependents	III-106
Making Changes That Require Demotion	III-107
Index	Index-1

How to Use This Book

The *Rational Environment User's Guide* is intended for use by both beginning and more experienced users:

- If you are a beginning user with no previous formal instruction on the Rational Environment™, you can use this book to teach yourself the basic concepts and operations for using the Environment.
- If you have completed the Fundamentals training course, you can use this book to reinforce and supplement what you have learned about the Environment with additional detail.

The User's Guide is similar in scope to the Fundamentals training course. Whereas the Fundamentals training course provides hands-on experience with the Environment through guided exercises, the User's Guide provides descriptions of Environment concepts illustrated with example scenarios and sample screens. You can use the User's Guide when no other experts are available to answer your questions.

The User's Guide can be used in conjunction with the *Rational Environment Basic Operations*. The Basic Operations is a quick reference that provides the specific steps for accomplishing a particular task. If you have questions on the tasks or steps listed in the Basic Operations, you can refer to the User's Guide. The User's Guide provides context for each task and supplements the steps with definitions of terms, explanatory material, and examples.

For a quick overview of the Rational Environment itself, see "Key Concepts: Rational Environment," following this section.

Organization of This Guide

The User's Guide presents information about the Environment in three parts.

Part I, "Getting Started," describes the following general features of the Environment:

- Using the keyboard
- Logging in
- Finding and displaying Environment objects

- Using multiple windows
- Executing commands
- Getting help

Part II, "Editing Text," describes how to create, edit, and save text files containing documentation, test data, and the like. Basic text editing operations are covered here; note that these are also used when editing Ada[®] units.

Part III, "Developing Simple Ada Programs," describes what you need to know to create, edit, execute, debug, browse, and modify Ada programs.

This guide often refers you to the *Rational Environment Reference Manual* for facilities not covered in this guide. The *Rational Environment Reference Manual* also gives detailed information about each Environment command.

What You Should Read

You can think of this book as a textbook resource on the Environment. As with any textbook, you can read through all the chapters sequentially, at your own pace.

For a more accelerated approach, you can read the table of Contents and then study only selected sections of immediate interest. Alternatively, you can scan each chapter, reading in detail where desired. Your familiarity with other computer systems should help you decide which sections you need to read and which details you can skip.

For example, to start developing an Ada program right away, you can scan the chapters in Parts I and II and then read the following chapter in detail:

- "Overview of Ada Unit Development," Chapter 9.

This chapter summarizes the basic information about creating, editing, compiling, and executing Ada units. (More information about these issues is available in Chapters 10 through 12.) Also recommended for detailed reading are:

- "Debugging Ada Programs," Chapter 13.
- "Browsing Ada Programs," Chapter 14.
- "Modifying Installed or Coded Ada Programs," Chapter 15.

Key Concepts: Rational Environment

The Rational Environment is a software development environment that provides highly integrated facilities for designing, implementing, debugging, and maintaining programs written in Ada. In addition, the Environment provides facilities for managing the design, decomposition, and implementation of large software projects.

At a very general level, the bulk of your work on the Rational Environment consists of *editing* various kinds of *objects*. Objects are structured representations of various kinds of information; editing means applying various operations to view or modify the form or content of objects. The basic types of Environment objects include:

- *Files*, which can contain documentation, test data, and the like.
- *Ada compilation units*, which contain Ada code represented in an underlying structure distinct from files. This underlying structure embodies the syntactic and semantic constructs of the code.
- *Libraries*, which are analogous to directories on other computer systems. Libraries contain objects such as files, Ada units, and other libraries.

The Environment provides a consistent set of operations for editing the different types of objects. Basic editing operations allow you to create, view, modify, preserve, and delete objects or elements within objects. Additional kinds of operations are available where appropriate.

Though distinct in important ways, text files and Ada units have a number of features in common. Objects of both kinds can be opened for editing, modified with text editing operations, and saved. Multiple versions can be retained. Objects in use are protected by write locks, which prevent modification by other users.

Powerful editing operations are available specifically for Ada units. These operations take advantage of the information in the underlying representation of Ada units. Important Ada-specific operations include:

- Pretty-printing, syntactic completion, and error checking in source code
- Semantic error checking in source code
- Generation of skeletal package bodies from package specifications and generation of skeletal private parts from private type declarations

Libraries form the basis for the hierarchic organization among Environment objects. The root of this hierarchy is the library ! (pronounced “bang”). The library hierarchy forms the basis for naming Environment objects. For example, !Users.Anderson names a library called Anderson contained in a library called Users in the root library. Each user with an account on the Environment has a home library in !Users that is named with his or her username. Thus, !Users.Anderson can be the home library of a user whose username is Anderson.

You can *traverse* the library hierarchy to view the various objects in it. Traversing to an object in the library hierarchy displays that object and makes it the current editing context.

When you traverse to an Environment object, it is displayed in a window on the screen. You can have multiple windows, which you can scroll, enlarge, shrink, join, transpose, remove from the screen, and redisplay.

The basic way to communicate with the Environment is by entering commands. Through Environment commands, you can perform traversal operations, editing operations, window management operations, and the like. A command can be executed in one of two ways:

- By pressing a key or combination of keys to which the command is bound
- By opening a special window, called a Command window, in which you can enter and then *promote* the command and its arguments

The Environment’s command language is Ada. That is, the commands and tools provided by the Environment are Ada procedures and functions. These subprograms are defined in packages, whose names reflect the various Environment objects and other functional groupings of Environment operations. The specifications of these packages are located in the libraries !Commands and !Tools; you can view the package specifications by traversing to them.

Because Environment commands are written in Ada:

- Command naming and syntax follow Ada rules.
- The same interfaces for entering commands (Command windows or key bindings) can be used for executing your own Ada subprograms.

Under the standard key bindings provided by the Environment, the most frequently used commands are bound to keys or combinations of keys. (You can change these bindings to suit your own needs.) Your keyboard has an overlay that lists the various operations you can perform with particular key combinations. You can think of this overlay as an analog to pull-down menus on other computer systems— from the overlay, you can locate the operation you want to perform and then invoke it with the indicated key combination.

In general, operations that act on objects (or elements within objects) involve two steps:

1. Pointing to (*designating*) the object or its representation on the screen
2. Requesting the appropriate action, either by pressing a key combination or entering a command

Some operations allow you to designate an object through cursor position; other operations require that you *select* the object so that it appears in a highlighted font. Designation is analogous to using a mouse on other computer systems and provides an alternative to entering full object names as arguments to commands.

On-line help is provided for each Environment command. Requesting help on a command results in a display of a description of the command and its parameters. The help facility also provides information about key bindings.

When you compile a program on the Environment, the compilation order is determined automatically from the dependencies among program units. In addition to batch compilation tools, the Environment provides an interactive compilation system that breaks the compilation process into phases. Under this system, compiling a unit entails *promoting* the unit through a series of states, one for each phase. The generation of object code for execution defines the final state in the series. Thus, errors can be checked and dependencies verified at intermediate states without waiting for complete compilation.

In contrast to batch compilation on other computer systems, preparing a unit for execution on the Environment does not produce additional objects such as object files or executable images. Therefore, you are ensured that the program you are executing always matches its source code.

You can debug Ada programs interactively with the source-level Rational Debugger. Using multiple windows, the Debugger tracks an executing program's progress in the program source. With "point and act" operations as described above, you can set breakpoints, query variable values, inspect stack frames, and the like.

You can browse systems of Ada units by pointing to an Ada identifier and asking to see its defining occurrence. Furthermore, you can point to an Ada identifier and ask to see its using occurrences within a single unit or across all units on the system. When you browse an Ada system, the Environment follows the dependencies introduced by *with* clauses, so that it can display the appropriate Ada units no matter what libraries they are in.

Upward-compatible changes can be made in compiled programs without requiring the recompilation of dependent units. By using *incremental operations*, you can make incrementally compiled changes to statements, comments, and declarations that are not referenced elsewhere in the program.

RATIONAL

Part I. Getting Started

Contents

Chapter 1. The Rational Terminal Keyboard	I-1
The Rational Terminal Keyboard Layout	I-2
Main Keyboard	I-4
Function Keys	I-4
Numeric Keypad	I-4
Cursor Keys	I-5
Modifier Keys	I-5
Item Keys	I-5
Auxiliary Keys	I-6
Executing Key Combinations	I-6
Item-Operation Key Combinations	I-6
Executing an Item-Operation Key Combination	I-6
Modified Key Combinations	I-7
Executing a Modified Key Combination	I-7
The Rational Terminal Keyboard Overlay	I-8
How the Keyboard Overlay Is Organized	I-8
Reading the Keyboard Overlay	I-8
Summary of Key Notation	I-10
Symbols	I-10
Numeric Arguments	I-10
Case of Keys	I-11
Chapter 2. Logging In and Logging Out	I-13
Logging Into the Rational Environment	I-13
The Basic Login Process	I-13
Logging Into Nondefault Sessions	I-15

Logging Into Multiple Sessions	I-15
Checking the Terminal Type	I-16
Changing Your Password	I-17
Logging Out	I-18
Logging Out with Unsaved Changes	I-18
Chapter 3. Traversing the Rational Environment	I-21
What You See When You Log In	I-21
Message Window	I-22
Major Window	I-22
Home Library	I-22
Command Window	I-23
Customized Library Display Format	I-23
Customized Login Display	I-24
Moving between and within Windows	I-24
Objects in the Environment	I-25
Files	I-25
Ada Compilation Units	I-26
Libraries (Worlds and Directories)	I-27
Environment Objects and Access Control	I-28
The Environment Library Structure	I-28
Fully Qualified Object Names	I-30
The Current Context in the Library Hierarchy	I-30
Traversing the Environment Library Structure	I-31
Traversing from a Library to an Object in It	I-32
Traversing between Ada Specifications and Bodies	I-36
Returning Home	I-38
Traversing to the Enclosing Library	I-40
Traversing the Environment: Summary	I-42
Chapter 4. Managing Windows	I-45
Windows and Images	I-45
Scrolling an Image	I-45
Window Banners	I-47
Modification Symbols in the Window Banner	I-48
How Windows Are Placed on the Screen	I-49
Windows and Frames	I-49
Moving between Windows	I-50
Default Window Placement	I-51

Controlling Window Placement by Locking Windows	I-51
Controlling Window Placement through Traversal Commands	I-52
Redisplaying Windows Using the Window Directory	I-52
Displaying the Window Directory	I-53
Redisplaying Replaced Windows	I-54
Checking the Window Directory Before Logging Out	I-55
Changing Window Size and Placement	I-56
Joining Frames	I-56
Expanding Windows	I-57
Shrinking Windows	I-58
Making Frame Sizes Equal	I-58
Removing Windows from the Screen	I-58
Rearranging Windows on the Screen	I-59
Changing the Number of Frames	I-60
Chapter 5. Executing Commands	I-63
Using Command Windows	I-64
Unrecognized Commands	I-67
Correcting Typing Errors	I-68
Canceling Command Execution	I-68
Command Windows and Attached Windows	I-68
Ada Usage in Command Windows	I-69
Entering Parameters	I-69
Using the Command Window Declare Block	I-71
Getting Prompting Assistance	I-72
Getting Formatting Assistance through the Promote Key	I-72
Getting Semantic Completion through the Complete Key	I-72
Filling In Parameter Prompts	I-73
Moving between Prompts	I-73
Completing Ambiguous Name Fragments	I-74
Completion Menu Entries	I-75
Abbreviating Commands	I-76
Clearing a Command Window	I-76
Reusing Command Windows	I-77
Executing Subsequent Commands	I-77
Reexecuting the Same Command	I-77
Modifying and Reexecuting Commands	I-78
Recalling Previous Commands	I-79
Keys and Command Windows	I-79

Environment Commands	I-80
Fully Qualified Ada and Environment Names	I-80
Visibility in Command Windows	I-81
Special Names and Parameter Placeholders	I-81
Chapter 6. Getting Help	I-83
How Do the Help Keys Work?	I-83
What Command Does This Key Execute?	I-84
Reading Help Messages	I-84
What Does This Command Do?	I-85
What Commands Pertain to This Topic?	I-86
Reading Help Menus	I-87
Getting Further Help from a Menu	I-88

Chapter 1. The Rational Terminal Keyboard

Many Rational Environment commands are executed by pressing various keys and combinations of keys. Therefore, it is important for you to become acquainted with key usage in the Environment.

The *Rational Environment User's Guide* assumes that you are using the standard key bindings that are provided by the Rational Environment. As you become more familiar with the Rational Environment, you can tailor your key bindings to reflect your specific needs (see "Rebinding Keys," in the *Rational Environment Basic Operations*).

This guide additionally assumes that you are using a Rational Terminal, which has been customized for use with the Rational Environment. Therefore, examples throughout this guide are written using key names as they appear on the Rational Terminal keyboard and keyboard overlay. (Key names appear in small boxes—for example, Delete.)

If you are using a VT100™ terminal, a VT100-compatible terminal, or a Facit terminal, you can find equivalent standard key bindings listed in the Keymap (located in the Reference Summary, Volume 1 of the *Rational Environment Reference Manual*) and in the Basic Keymap (located in the *Rational Environment Basic Operations*).

This chapter describes:

- Where to find keys on the Rational Terminal keyboard
- How to use item keys and modifier keys in key combinations
- How to read the Rational Terminal keyboard overlay
- How to interpret the notation used to indicate keys and key combinations

For a more complete description of the Rational Terminal and its keyboard, see the *Rational Terminal User's Manual*.

The Rational Terminal Keyboard Layout

The overall layout of the Rational Terminal keyboard is shown in Figures 1-1 and 1-2.

As Figure 1-1 shows, the keys on the keyboard fall into seven functional groups, including:

Main keyboard	Modifier keys
Function keys	Item keys
Numeric keypad	Auxiliary keys
Cursor keys	

The following sections identify these groups of keys and briefly describe what the keys do when you are logged into the Rational Environment. (Later chapters contain more information about the usage of individual keys as appropriate.)

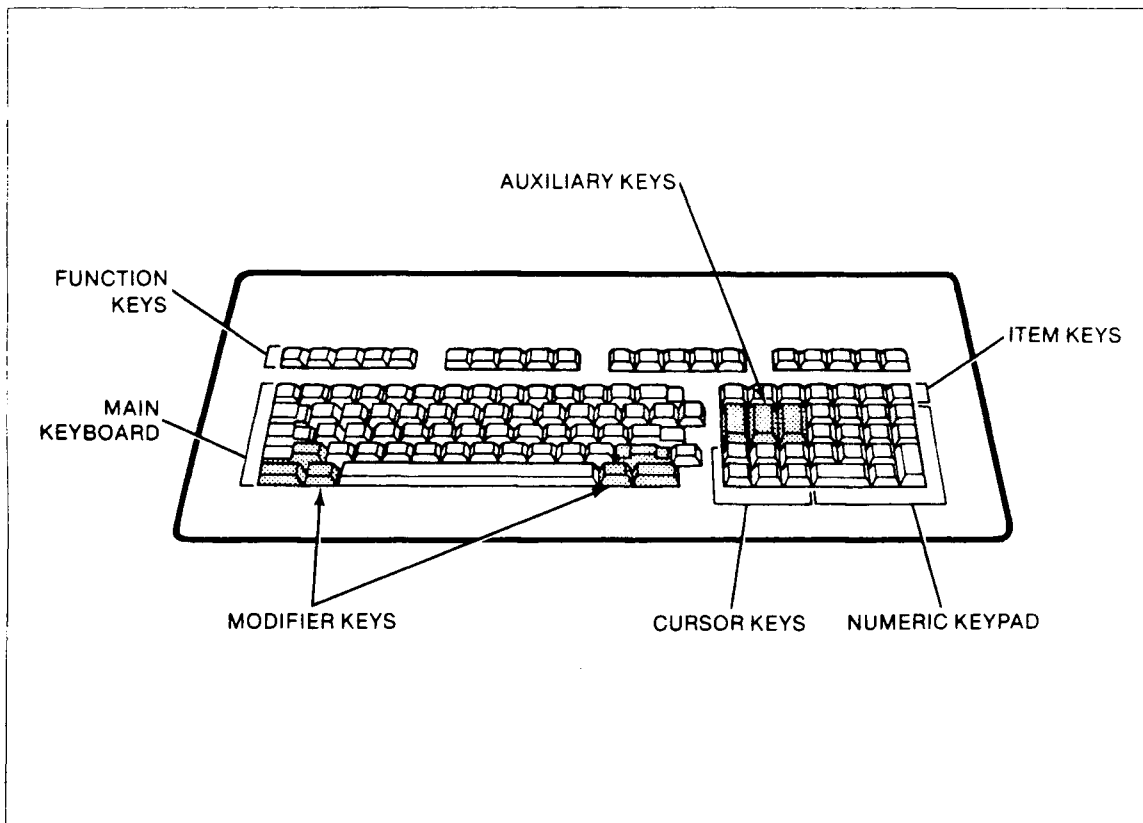


Figure 1-1. Groups of Keys on the Rational Terminal Keyboard

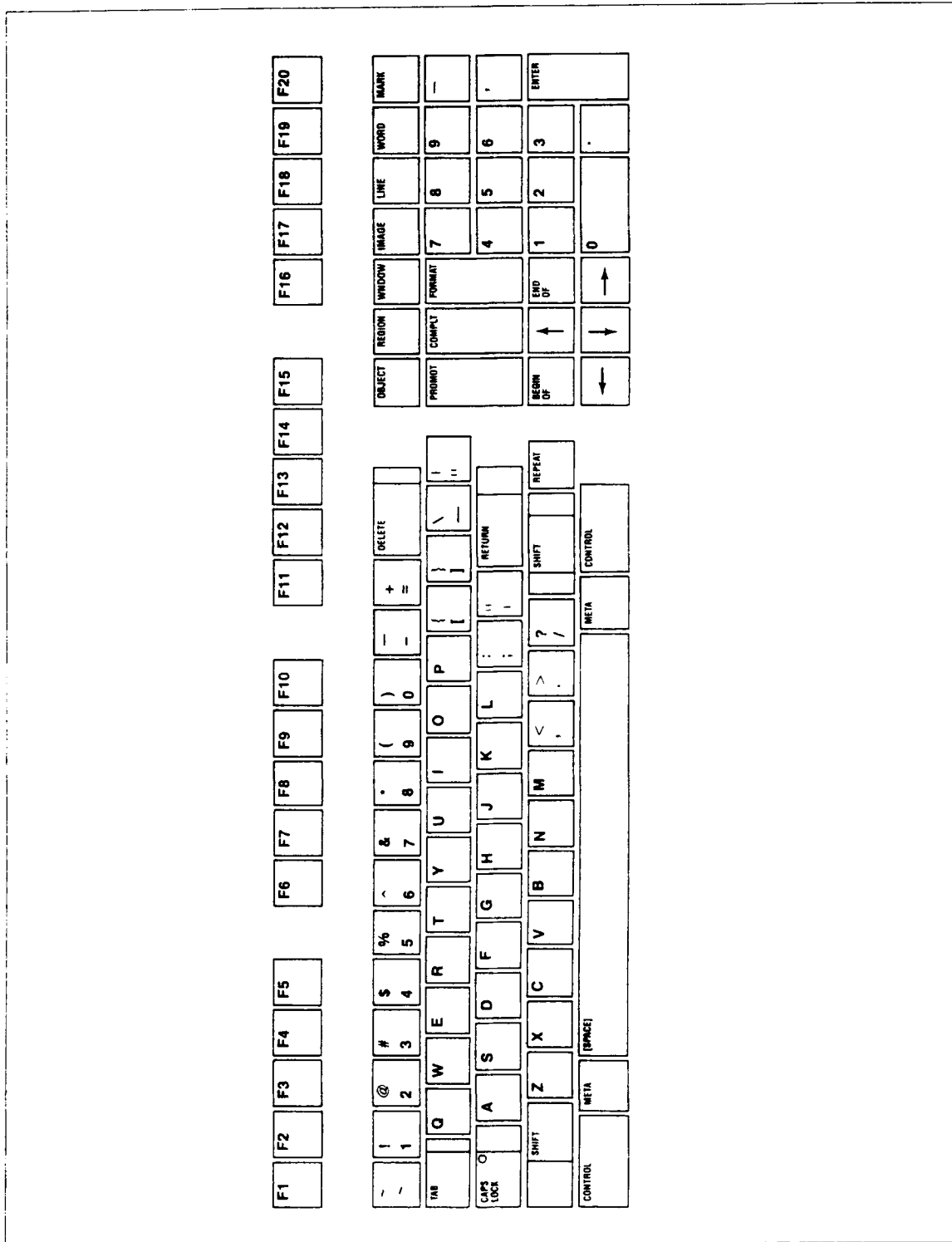


Figure 1-2. The Rational Terminal Keyboard Layout

Main Keyboard

The main keyboard contains all alphanumeric and punctuation keys in the standard ANSI typewriter keyboard layout.

Some particularly useful keys in this portion of the keyboard include:

- **Delete**, which deletes the character just to the left of the cursor.
- The unshifted underscore and unshifted double quote, which simplify the entry of Ada identifiers and quoted strings.
- **Return**, which enters a new line.
- **Repeat**, which, when held down along with another alphanumeric key, rapidly enters multiple instances of the alphanumeric character indicated by the other key. (Using **Repeat** is faster than simply holding down an alphanumeric key until the character repeats.)
- **Shift**, **Control**, and **Meta**, which are discussed under “Modifier Keys,” below.
- The space bar, which enters a blank character.

Function Keys

The function keys are a row of 20 keys at the top of the keyboard.

These keys are used alone or in combination with modifier keys (see “Modifier Keys,” below). Most function keys, whether used alone or in combination, are bound to specific Environment commands. You can bind the unbound keys and key combinations and you can change the existing bindings (see “Rebinding Keys,” in the *Rational Environment Basic Operations*).

Numeric Keypad

The numeric keypad is a block of 14 keys, including 10 number keys, **Enter**, and three keys with punctuation symbols: minus (**-**), comma (**,**), and period (**.**).

Under the standard key bindings, the number and punctuation keys on the numeric keypad are used very differently from the corresponding keys on the main keyboard:

- **Number keys**: The number keys on the numeric keypad enter numeric arguments for certain Environment commands bound to keys. For example, to move the cursor up three lines, press **3** on the numeric keypad, followed by **↑**.

In contrast, the number keys on the main keyboard enter numeric characters into text. For example, to enter numeric test data into a file, use the main keyboard number keys. (Do not use numeric keypad keys for this purpose.)

- **Minus key (-)**: The minus key on the numeric keypad indicates a negative value for a command argument.

In contrast, the minus key on the main keyboard enters a minus sign or hyphen character into text.

- Period key (`Ⓜ`): The period key on the numeric keypad sends an end-of-file terminator during interactive input.
In contrast, the period key on the main keyboard enters a period character (or decimal point) into text.
- Comma key (`Ⓝ`): The comma key on the numeric keypad converts a subsequent main keyboard number key into a numeric argument.
In contrast, the comma key on the main keyboard enters a comma character into text.

To distinguish numeric keypad keys from main keyboard number keys, you will see the following notation:

- The word “numeric” inside the box notation indicates a key on the numeric keypad—for example, `numeric 2` and `numeric -`.
- The names of main keyboard keys are simply boxed—for example, `3` and `,`.

Cursor Keys

The cursor keys are a group of six keys to the left of the numeric keypad.

The cursor keys can be used alone or in combination with modifier or item keys (see “Executing Key Combinations,” below). The cursor keys primarily move the cursor on the screen, although some combinations with the cursor keys execute other kinds of operations. The direction of cursor movement is indicated by the arrows on the arrow keys (`Ⓜ`, `Ⓝ`, `Ⓟ`, `Ⓡ`). `Begin Of` and `End Of` move the cursor directly to the beginning or end of an item such as a line of text.

Modifier Keys

The modifier keys include `Shift`, `Control`, and `Meta`. A set of three modifier keys is located in each of the two lower corners of the main keyboard.

Modifier keys are used only in combination with another key—for example, a key on the main keyboard, a function key, an auxiliary key, or a cursor key. By forming key combinations to which Environment commands can be bound, the modifier keys extend the use of the keyboard. See “Executing Key Combinations,” below.

Item Keys

The item keys are a row of seven keys above the numeric keypad. Item keys refer to items that you frequently need to reference during Environment editing operations: words, lines, regions, and so on. The item keys are shown in Figure 1-3.

Item keys are used only in combination with another key—for example, a key on the main keyboard, a function key, an auxiliary key, or a cursor key. See “Executing Key Combinations,” below.

Auxiliary Keys

The three auxiliary keys are located directly above the cursor keys and are bound to specific, frequently used Environment commands. Note that, although the key names are abbreviated on the actual keycaps, these key names are spelled out in the documentation notation: `Promote`, `Complete`, and `Format`.

Executing Key Combinations

Item and modifier keys (see above) are not used alone but are used only in combination with another key, such as a key on the main keyboard, a function key, an auxiliary key, or a cursor key. In this way, both item and modifier keys extend the use of the keyboard by providing for more key-binding possibilities. Note that the standard Environment key bindings are given in the Basic Keymap, in the *Rational Environment Basic Operations*; even more detail is given in the Keymap, found in the Reference Summary (Volume 1 of the *Rational Environment Reference Manual*).

A key combination consists of two or more sequential or overlapping keystrokes, depending on whether the first keystroke is an item key or a modifier key.

Item-Operation Key Combinations

A key combination that contains an item key (such as `Word`, `Line`, and the like) is typically bound to a command that operates on the item named by the item key. The second keystroke in such a combination is called the *operation* key, because it identifies the action that applies to the indicated item. This combination is referred to as an *item-operation combination*.

Operation keys can be main keyboard keys, function keys, auxiliary keys, or cursor keys. In general, commands that execute similar operations are bound to combinations containing a common operation key. For example, the combinations `Line` - `D`, `Word` - `D`, and `Window` - `D` all delete an item, as indicated by the shared operation key, `D`.

Executing an Item-Operation Key Combination

The keystrokes must be sequential in an item-operation key combination. The item key acts as a command prefix, indicating that the following keystroke should be interpreted as a command and not as a character to be inserted. Item-operation combinations are executed as follows:

1. Press and release the item key.
2. Press and release the operation key.

In the notation for an item-operation combination, the two keys are separated with a hyphen to remind you that the keystrokes are sequential: `item key` - `operation key`.

Note that once you have pressed an item key, the next key you press is interpreted as part of a key combination, no matter how long you wait before pressing the next key. In other words, an item key does not “expire” after a delay. In fact, if you wait a few seconds after pressing an item key, the item key name is displayed at the top of the screen as a reminder.

If you unintentionally press the wrong item key, you can cancel it by pressing the `Control``G` key combination (see “Modified Key Combinations,” below). Alternatively, you can supply a second keystroke that forms an unbound combination, so that no command is executed. Since, in the standard key bindings, no commands are bound to key combinations containing two item keys, you can cancel a given item key by pressing it a second time.

Modified Key Combinations

A key combination that contains one or more modifier keys (`Shift`, `Control`, or `Meta`) is a modified key combination. Any combination of `Shift`, `Control`, or `Meta` can modify keys on the main keyboard, function keys, cursor keys, or auxiliary keys. However, modifier keys are not used in combination with item keys.

Note that an alphanumeric key modified by `Shift` inserts either a capital letter or punctuation. Furthermore, when an alphanumeric key is modified by `Control` or `Meta`, the use of `Shift` is generally ignored. (That is, `Control``Shift``D` is equivalent to `Control``D` in the standard key bindings.)

Executing a Modified Key Combination

The keystrokes must overlap in a modified key combination. That is, the combination is executed as follows:

1. Press and hold the modifier key or keys.
2. While holding down the modifier key(s), press the key to be modified.

In the notation for a modified key combination, the two keys are shown without a hyphen to remind you that the keystrokes overlap: `modifier key``function key` or `modifier key``alphanumeric key`.

The use of different combinations of modifiers allows a family of related commands to be bound to a common modified key. For example, a family of commands that provides system help is bound to `F11` and its modified key combinations:

- `F11` displays help on selected Environment objects.
- `Shift``F11` displays an explanation of the help facility.
- `Meta``F11` displays help on keys.
- `Control``F11` moves the cursor into a Help window.

The Rational Terminal Keyboard Overlay

In the standard key bindings for the Rational Environment, many commands are bound to modified function keys. These commands, and the key combinations to which they are bound, are indicated on the Rational Terminal keyboard overlay that is provided with your Rational Terminal.

The Rational Terminal keyboard overlay fits over the function keys, covering the upper portion of the keyboard. The overlay has two transparent pockets that contain removable cards on which the standard key bindings are printed. (If you change your key bindings, you can remove the cards, write the new bindings on them, and reinsert the cards into the overlay pockets. You can also write directly on the overlay, and the back can be used for your special applications.) The keyboard overlay is shown in Figure 1-3.

How the Keyboard Overlay Is Organized

The information on the keyboard overlay is organized in columns, one for each function key that has commands bound to it. At the top of each column, a single word indicates the family of Environment commands bound to the corresponding key and to its modified key combinations. For example, note in Figure 1-3 that the commands bound to **F11** provide help, the commands bound to **F15** create things, and the commands bound to **F6**, **F7**, **F8**, and **F9** are used in debugging.

The entries within each column indicate the specific commands that are bound within each command family. Note that the column entries are abbreviated forms of the corresponding command names, not the actual command names themselves. (You can find the actual command names using the help facility; see Chapter 6, "Getting Help.")

Within the columns, the command entries are aligned in eight rows. The bottom row corresponds to the unmodified use of each function key. The next seven rows correspond to the seven possible combinations of modifier keys—namely, **Shift**, **Control**, **Meta**, **Control|Shift**, **Control|Meta**, **Meta|Shift**, and **Control|Meta|Shift**. The seven modifier combinations are printed in red in the center and on either side of your keyboard overlay. Therefore, each command entry on the keyboard overlay labels a unique combination of zero or more modifiers and a function key. (Note that a blank entry in a column indicates that a particular combination has not been bound.)

Reading the Keyboard Overlay

At this point, it is clear that a given modified function key combination can be referred to in either of two ways:

- Using the actual key names that appear on the keycaps—for example, **Meta|F11**
- Using the logical key names that appear as entries on the keyboard overlay—for example, **Help On Key**

					CONTROL META SHIFT META SHIFT	Debug Task Display	Debug Show Breaks	Debug Rendezvous Info	Debug Debugger Window	Traverse Enclosing Library
					CONTROL META	Stop	Remove Breaks	Show Exceptions	Modify	Enclosing In Place
					CONTROL SHIFT					Home Library
					META	Run Returned	Activate	Propagate	Set Pointer Level	Other Part In Place
					CONTROL	Run Local	Break	Catch	Set Element Count	Enclosing In Place
					SHIFT	Execute	Break ~Default	Forget	Set First Element	Other Part
						Run	Show Source	Stack	Put	Definition In Place
										Definition
F1	F2	F3	F4	F5		F6	F7	F8	F9	F10
Help	CMVC Checked Out In View	Promote Code (All Worlds)	Demote Source (All Worlds)	Create Create Private	CONTROL META SHIFT META SHIFT	Show Show Usage (Indirect)	Misc. Make Inline	Items Make Separate	Jobs	Info. Show Access List
	Checked Out By User	Install (All Worlds)	Unicode (All Worlds)							
Print	Accept Changes	Code (This World)	Source (This World)	Create Text	CONTROL META	Show Usage	Show Unused	Previous Prompt	End Of Input	What Object
	Show Info		Unicode (This World)	Create World	CONTROL SHIFT			Previous Underline		
Help On Key	Check In	Install (This World)	Withdraw Unit	Create Directory	META	Show Usage (Unit)	Show Unused (Unit)	Next Prompt	Job Connect	What Locks
Help Window	Check Out	Code Unit	Source Unit	Create Ada	CONTROL	Underlines Off	Item Off	Previous Item	Job Kill	What Users
Help On Help	Show Out of Date	Install Stub	Demote	Create Body	SHIFT	Show Errors		Next Underline	Job Enable	What Load
Help	Prompt For	Install Unit	Edit	Create Command		Semanticize	Explain	Next Item	Job Disable	What Time
F11	F12	F13	F14	F15		F16	F17	F18	F19	F20

Figure 1-3. The Rational Terminal Keyboard Overlay

Examples throughout the Rational Environment documentation refer to modified function keys by using the logical key names, since these are more meaningful than actual keycap names like `Control` and `F11`. Furthermore, the logical key names from the Rational Terminal keyboard overlay are also used on overlays for VT100-compatible or Facit terminal keyboards, where the actual key combinations may be different.

When an example refers to a key combination by its logical name, you can use the keyboard overlay to find the actual keys to press. For example, to find the key combination corresponding to `Job Kill`:

1. Locate the logical key name "Job Kill" on the keyboard overlay. (Use the column headings—in this case, "Jobs"—to help you narrow down the search to the appropriate topic area.)
2. Note the function key that corresponds to the column that contains the logical key name. In this example, "Job Kill" is in the column corresponding to `F19`.
3. Note the modifier key combination that corresponds to the row that contains the logical key name. In this example, "Job Kill" is in the row corresponding to `Control`.
4. Use the combination `ControlF19` when instructed to press `Job Kill`.

Summary of Key Notation

The following notational conventions are used in this *Rational Environment User's Guide* and in all other Rational Environment documentation.

Symbols

<code>key name</code>	Press the key with "key name" on its keycap.
<code>key name</code>	Press the key or key combination designated by "key name" on the overlay.
<code>key1</code> - <code>key2</code>	Press and release <code>key1</code> ; then press <code>key2</code> .
<code>key1</code> <code>key2</code>	Press and hold <code>key1</code> while pressing <code>key2</code> .
<code>numeric 1</code>	Press <code>1</code> on the numeric keypad.

Numeric Arguments

You can give numeric arguments to many Environment commands that are bound to keys. To do so:

1. Press and release the appropriate number key on the numeric keypad.
2. Press the key combination bound to the desired command.

For example, `Word` - `D` deletes one word. The following combination deletes four words: `numeric 4` - `Word` - `D`.

You can indicate negative numbers by pressing and releasing `numeric -` first. For example, the following combination shrinks a window by seven lines (“expands” it by -7 lines):

`numeric - - numeric 7 - Window - I`

Case of Keys

Although keys are shown in the documentation as uppercase, the unshifted equivalent also works. This is true for the nonalphabetic characters as well. For example, `Object - d` is equivalent to `Object - D` and `Object - i` is equivalent to `Object - I`.

RATIONAL

Chapter 2. Logging In and Logging Out

This chapter describes how to:

- Log into the Rational Environment
- Log into multiple sessions
- Change your password
- Log out

Logging Into the Rational Environment

You must have access to a user account in the Rational Environment in order to log in. With a user account, you are identified to the Environment through a username and a password. If you do not have a valid username and password, see your system manager.

You can log into the Rational Environment from a Rational Terminal, a VT100 terminal, a VT100-compatible terminal, or a Facit terminal. Logging in follows the same steps, regardless of the terminal type.

The Basic Login Process

To log in, you must respond to a series of login prompts. Note that:

- You can enter login information in uppercase or lowercase letters.
- You must press `Return` to complete each response and display the next prompt.
- If you discover an error before you press `Return`, you can press `Delete` to erase individual characters, or you can press `Control|U` to erase your entire input for that prompt.

To log in:

1. Make sure your terminal is turned on. Your terminal displays the following prompt when it is connected to the Rational Environment and ready for you to log in:

Commence Login

2. Press **[Return]** to display the prompt that requests a username:

Enter user name:

3. Enter your username at the prompt and press **[Return]**.

The system responds with a prompt that requests a password:

Enter password:

4. Enter your password at the prompt and press **[Return]**. Note that your password does not appear on the screen as you type it.

- If your username and password are accepted, the system responds with a prompt that requests a session name:

Enter session name:

Sessions provide a means by which you can customize certain aspects of your work environment for individual projects. Associated with each account is a default session, called S_1, along with any number of additional user-created sessions.

- If your username and password are not accepted, the system responds with the following message:

Incorrect user name or password.

Commence Login

Verify your username and password, and then repeat steps 2–4 to try again.

5. If you are a new user or if you do not require a customized session, log into the default session, S_1. To do this, press **[Return]** at the session prompt.

(See also “Logging Into Nondefault Sessions,” below.)

When you have logged in successfully:

- The Environment displays a message indicating the last time you were logged into the specified session (there is no message for new sessions).
- The screen briefly goes blank.
- A login procedure is executed, which causes your Environment session to appear on the screen. The next chapter explains what you see on the screen at this point.

Logging Into Nondefault Sessions

As indicated by steps 4 and 5 above, logging into your Environment user account entails logging into a specific session that is associated with your account. You can always log into the default session, S_1 (as in step 5). Furthermore, whenever you log in, you have the option of creating a new session or logging into a previously created session. The main advantage of having multiple sessions is that each session can serve as an individual work environment that you can set up specifically for a particular ongoing activity. (For more details about sessions and customizing the Environment, see the Session and Job Management (SJM) book of the *Rational Environment Reference Manual*.)

To create a new session or log into an existing nondefault session:

1. Follow steps 1 through 4 of the basic login process, above.
2. When the session prompt is displayed, enter a name for the new session. A session name can be any Ada identifier—for example, S_2. Then press :

```
Enter session name: s_2 
```

3. If you specify an existing session name, the login process continues as described above.

If you specify a new session name, the following prompt asks you to verify that you want to create that session:

```
<name> does not exist. Do you want it created? (Y or return => Yes):
```

- Press to create the session and continue the login process.
- If you made an error specifying the session name, enter n to redisplay the Enter session name: prompt. You do not need to press . Continue with step 2.

Once a session is created, it exists in your home library even when you are not logged into it.

Logging Into Multiple Sessions

Besides allowing you to create and preserve customized editing environments for individual projects, sessions also provide a means for you to log in concurrently from several ports. For example, if you are logged into your default session, S_1, and you want to log in a second time on a different terminal, you can create a second session—say, S_2—for this purpose.

Note that you cannot log into the same session twice. If you try, the following message is displayed:

```
Last login on July 1, 1987 at 3:17:73 PM
<name> is in use. Choose another session or try again later.
```

```
Try another session? (Y or return => Yes):
```

At this point, you can:

- Press `[Return]` to display the Enter session name: prompt and continue the login process.
- Enter `n` to cancel the login and redisplay the Commence Login prompt. You do not need to press `[Return]`.

Checking the Terminal Type

The Rational Environment automatically checks the terminal you are using and adjusts the port to the correct terminal type. If the Environment cannot determine the correct terminal type, the following prompt is displayed automatically after you have specified the session name:

```
Enter terminal type:
```

If you want to ensure that the port's terminal type is correct before you try to log in, you can display and adjust the terminal type yourself. However, this procedure normally is not necessary.

To display and adjust the terminal type:

1. At either the Commence Login or the Enter user name: prompt, enter an equals sign (=) and press `[Return]`:

```
Enter user name: = [Return]
```

The system responds by displaying the Enter terminal type: prompt with the current terminal type for the port you to which you are connected:

```
Enter terminal type (RATIONAL):
```

2. If the terminal you are using matches the terminal type given in the prompt, press `[Return]`. Continue with step 3.

If the type of terminal you are using differs from the type displayed in the prompt, enter the correct type (for example, Facit or VT100) and press `[Return]`. The port retains the new terminal type until you change it again.

3. The Enter user name: prompt is redisplayed. Log in as usual.

In the unlikely event that you have logged into a terminal that does not match the port's terminal type (for example, the automatic adjustment has failed), keystrokes will not be interpreted correctly and you will not be able to log out. See your system manager.

Changing Your Password

At many installations, new users are given temporary passwords when their accounts are created. It is then each user's responsibility to change his or her password to ensure the security of the account.

To change your password:

1. Once you have logged in and the Environment is displayed, press `Create Command` to open a window in which you can enter a command. (This window is called a *Command window*.)
2. In the Command window, enter the following command, where the values of `string1`, `string2`, and `string3` are explained below. Be sure to enclose each string in double quotation marks and to separate the three parameters with commas:

```
operator.change_password("string1","string2","string3");
```

`string1` Your username.

`string2` The current password for the account. If this password is not known, see your system manager.

`string3` The new password for the account.

3. Press `Promote`.

When the command has executed, it is displayed in reverse video in the Command window. Use your new password the next time you log in.

For example, a user named Anderson whose old password is "temp" enters the following command to change the password to "hello":

```
operator.change_password("anderson","temp","hello");
```

For details about entering Environment commands, see Chapter 5, "Executing Commands."

Logging Out

To log out of each **Environment** session:

1. Press `Create Command` to open a Command window in which you can enter a command.
2. In this window, enter the following command:


```
quit;
```
3. Press `Promote`.

For details about entering Environment commands, see Chapter 5, “Executing Commands.”

The Quit command executes if there are no unsaved editing changes and if there are no executing jobs (programs or commands) that are reading input interactively:

- The command is displayed in reverse video.
- The screen goes blank.
- The Commence Login prompt is displayed on the screen.

If there are editing changes that have not been saved, or if there are running jobs reading input interactively, the Quit command fails and a message is displayed.

At this point, you can save uncommitted editing changes, terminate pending jobs, and then repeat steps 1 through 3 to log out. (See “Checking the Window Directory Before Logging Out,” in Chapter 4.) Alternatively, you can log out, discarding unsaved changes. (See “Logging Out with Unsaved Changes,” below.)

Note that if you are logged into multiple concurrent sessions, you must log out of each one individually.

Logging Out with Unsaved Changes

You can log out with editing changes that have not been saved or with jobs (executing commands or programs) that are reading interactive input. If you do so, unsaved changes are discarded and jobs wait indefinitely for input.

To log out regardless of unsaved changes or pending jobs:

1. Press `Create Command` to open a Command window in which you can enter a command.
2. In this window, enter the following command:


```
quit(true);
```
3. Press `Promote`.

The `Quit` command with the parameter `true` executes without warning you if there are unsaved editing changes or pending jobs:

- The command is displayed in reverse video.
- The screen goes blank.
- The `Commence Login` prompt is displayed on the screen.

RATIONAL

Chapter 3. Traversing the Rational Environment

This chapter describes:

- The basic layout of the screen as it appears when you log in
- An overview of several important kinds of objects in the Environment
- An overview of the Environment directory structure
- The basic method for viewing elements in the directory structure

What You See When You Log In

After you have logged in, your screen will look similar to Figure 3-1:

```
Rational Environment
  D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
Rational (DelCo) ANDERSON S_1
-----
[Users Anderson : Library (World):
Calculation : ! Ada (Pack_Spec);
Calculation : ! Ada (Pack_Body);
Documentation : Library (Directory);
Factorial : S Ada (Func_Spec);
Factorial : S Ada (Func_Body);
Login : C Ada (Proc_Spec);
Login : C Ada (Proc_Body);
Memo_12_08_86 : File (Text);
My_Test_Data : File (Binary);
S_1 : Session;
S_1_Switches : File (Switch);
Tools : Library (World);
-----
RATIONAL ANDERSON library World
[Statement]
end;
```

Figure 3-1. A Typical Screen at Login

The screen in Figure 3-1 shows the Environment display for a user called Anderson. This screen contains several kinds of *windows* in which various kinds of information are displayed, such as system messages, a list of Environment objects belonging to the user Anderson, and a prompt for entering Environment commands. The following sections describe the basic types of Environment windows and what they contain when you log in.

Message Window

At the top of the screen is the *Message window*, which displays system status information, error messages, messages from the operator, and informative messages from jobs. The Message window in Figure 3-1 displays the messages that appear whenever you log in. The Message window retains all messages received since logging in, and you can scroll through this window to redisplay earlier messages.

The lower edge of the Message window is marked by a *banner*, which is a heavy, reverse video line containing information about the associated window. The Message window banner displays the name of your installation's R1000[®], your username, and the name of the session you are logged into. In addition, whenever you execute a command or run a job in the foreground, the Message window banner indicates this by displaying the `...running` message.

Major Window

Directly below the Message window and its banner is a *major window* containing your home library (which is described in the next section). Major windows are typically larger than other types of windows, for it is in major windows that you view and edit Environment objects such as Ada units, text files, library listings, and job output. A major window is automatically opened whenever you request to view or edit such an object. Since the screen can contain several major windows (the default number is 3), you can view several objects at a time.

As with the Message window, the lower edge of a major window is marked by a banner. This banner displays the name and other information about the object that is displayed in the window. The format and contents of major window banners are further described in "Window Banners," in Chapter 4.

Home Library

When you log in, a list of your Environment objects is displayed automatically in a major window. In general, such a list is called a *library*. Libraries are where you collect and organize your work. They are analogous to directories on other systems you may have used.

The particular library displayed at login is your *home library*. Your home library is associated with your user account and is named using your username. For example, in Figure 3-1, the name of the home library for user Anderson is `!Users.Anderson` and is displayed in the major window's banner. The library's name also appears underlined in the first line in the window.

A library lists an entry for each object it contains. In the example, the library `!Users.Anderson` already contains a number of objects, including files, Ada units,

and login sessions. The first time you log in, your home library will be empty except for listing your login session, S_1. As you create new objects, your home library will list more entries.

(See “Objects in the Environment,” later in this chapter, for more information on libraries and what they contain.)

Command Window

Below the major window and its banner, Figure 3-1 shows a two-line *Command window*. Command windows are where you enter Environment commands for execution. At login, a Command window is created automatically and the cursor is placed in it, ready for you to enter a command. You can enter successive commands in a single Command window, or you can create additional Command windows using `[Create Command]`. (Note that “Changing Your Password” and “Logging Out,” in Chapter 2, illustrated the creation and use of Command windows.)

Command windows are typically attached to major windows (as in this example), but they can also be attached to the Message window or even to other Command windows. Command windows are fully described in Chapter 5.

Customized Library Display Format

Figure 3-1 illustrates how user Anderson’s screen looks at login. This screen may differ from yours with respect to the amount of information that is displayed about each entry in libraries such as your home library. This and other aspects of the screen layout (such as the size of the Message window) are determined by *session switches*, which you can set to customize many features of your work environment; see the Session and Job Management (SJM) book of the *Rational Environment Reference Manual*.

The examples in the *Rational Environment User’s Guide* use the library display format that is shown in Figure 3-1. You may find the examples easier to follow if your library display format matches the format used in this guide. To make your format match this one, you can do one of two things:

- You can adjust the display for an individual library by placing your cursor in the library and pressing `[Explain]`. As you display multiple libraries, you can repeat this process for each library, as desired.
- You can ask your system manager to change the following switches to the given settings so that the next time you log in, all libraries will automatically match the display layout in this guide:

```
Library_Show_Standard      := True
Library_Std_Show_Unit_State := True
```

The second of these alternatives is the most convenient; however, it may reduce performance when viewing large libraries. It is recommended that you choose this alternative while learning to use the Environment and then restore the two switches to their default values when you become a more experienced user.

Customised Login Display

Each time you log in, an Ada procedure is executed to set up your session and determine exactly what appears on your screen. For new accounts, the system-wide default login procedure is used, which displays your home library, creates a Command window, and displays a message of the day, if there is one. It is possible, however, to create your own customized login procedure, which determines the windows that automatically appear on the screen at login. For example, as you work on different projects, you can have these displayed at login instead of your home library. For details about creating a customized login procedure, see Chapter 16 in the *Rational Environment Basic Operations*.

Moving between and within Windows

When you log in, the cursor is automatically in the Command window. You can move the cursor to the other windows on your screen as follows:

- Press `Window` - `↑` to move the cursor from the window it is in to the window above it.
- Press `Window` - `↓` to move the cursor from the window it is in to the window below it.

You can move several windows at a time by pressing a number on the numeric keypad first—for example, `numeric 2` - `Window` - `↑`. Further window management operations are fully described in Chapter 4.

Once the cursor is in a window, you can use the cursor keys to move the cursor up, down, left, or right.

Finally, you can scroll the contents of a window—for example, you can scroll up to see earlier messages in the Message window, or you can scroll down to see further entries in your home library:

- Make sure the cursor is in the window you want to scroll.
- Press `Image` - `↑` to scroll up. Alternatively, you can scroll up by moving the cursor up until you reach the upper window boundary.
- Press `Image` - `↓` to scroll down. Alternatively, you can scroll down by moving the cursor down until you reach the lower window boundary.

Note that for many item-operation key combinations, such as `Image` - `↑` and `Image` - `↓`, there are alternative “accelerated” key combinations, such as `Control` `Z` and `Control` `V`. Experienced users find accelerated key combinations somewhat more convenient to use. The Basic Keymap, in the *Rational Environment Basic Operations*, lists the accelerated alternatives to the key bindings given in this guide; for even more detail, see the Keymap in the Reference Summary, in Volume 1 of the *Rational Environment Reference Manual*.

Objects in the Environment

From the previous section, recall that major windows (like the one displayed at login) are where you view and edit Environment objects such as your home library. This section briefly describes the basic kinds, or *classes*, of Environment objects that you will work with most frequently. These include:

- Files
- Ada compilation units
- Libraries (more specifically, worlds and directories)

Note that there are other classes of objects in the Environment—for example, login sessions—many of which are introduced in other chapters. For now, the basic objects you need to know about are the ones described in the following sections.

Files

Environment files, like files on other computer systems, contain information. For example, files can contain binary test data, text for documentation, and the like. Different kinds of information are stored in different *subclasses* of files. Accordingly, documentation is stored in *text files* (files of subclass text) and binary test data are stored in *binary files* (files of subclass binary).

For the most part, the subclass of a file is established automatically by the Environment command that creates it. For example, the Text.Create command creates files of subclass text. Therefore, explicitly setting or changing a file's subclass, although possible, normally is not necessary. Note that the subclass of a file is set as an intrinsic attribute of the file and so does not depend on filenaming conventions such as filename extensions or suffixes.

In Figure 3-1, the library !Users.Anderson contains several entries for files. These entries indicate the files' subclasses in parentheses. For example:

```
Memo_12_08_86 : File (Text);
My_Test_Data  : File (Binary);
```

In such a display, the subclass serves as a useful reminder of what the file contains.

File subclasses like text and binary are predefined in the Environment. Other file subclasses that you are likely to encounter are *switch files*, which contain switches for tailoring various aspects of the Environment, and *activity files*, which pertain to large-system development using Rational Subsystems™. The Environment provides specialized commands for editing certain subclasses of files, such as text files and switch files.

Of the various subclasses of files, this guide covers only text files (see Part II, "Editing Text"). Switch files, activity files, and the Environment resources for creating binary files are covered in the *Rational Environment Reference Manual*.

Ada Compilation Units

On other computer systems that you may have used, Ada programs are written as text in files. In contrast, Environment files are not intended for use in creating and editing Ada programs (for example, there is no predefined file subclass for Ada programs). Rather, Ada compilation units such as packages, subprograms, and subunits constitute a separate class of Environment objects, for they are stored in data structures that have much richer representations than files. This richer underlying representation enables the Environment to provide features such as the following:

- Editing operations that act on the specific structure of the Ada programs you write—for example, syntactic formatting and completion
- A compilation management system that determines which units need to be compiled in a given Ada system and determines the correct compilation order
- A source-level debugger

Part III of this guide covers the various Environment facilities for developing Ada systems.

If necessary, Environment Ada units can be converted or copied into text files—for example, when transferring Ada programs to other computer systems or when including lines of Ada code in documentation. Furthermore, Ada programs that are in files can be parsed into Environment Ada units, as when transferring Ada programs from another computer system. However, as long as an Ada program is stored as a file, none of the Environment Ada editing facilities are available for it. Therefore, unless otherwise indicated, the term “Ada compilation unit” (or “Ada unit”) in this guide refers to an Environment Ada object, not to Ada source code written in a file.

The class of Ada compilation units has subclasses—one for each type of Ada compilation unit, as defined by the *Reference Manual for the Ada Programming Language*. Ada compilation units include the following:

- Procedure specifications and bodies
- Function specifications and bodies
- Package specifications and bodies
- Generic specifications and bodies
- Generic instantiations
- Subunits (including package, procedure, function, generic, and task bodies)

Figure 3-1 shows several entries for Ada compilation units in the library !Users.Anderson, including the specifications and bodies for the Login procedure, the Factorial function, and package Calculation. These entries indicate the units' subclasses using the abbreviated notation shown in parentheses:

```

Calculation      : C Ada (Pack_Spec)
Calculation      : C Ada (Pack_Body)

Factorial        : S Ada (Func_Spec)
Factorial        : S Ada (Func_Body)

Login            : C Ada (Proc_Spec)
Login            : I Ada (Proc_Body)

```

Note that entries for an Ada unit also contain a single letter (C, I, or S) that indicates the unit's *compilation state*. The letter C stands for the *coded* state, the letter I stands for the *installed* state, and the letter S stands for the *source* state. Ada unit compilation states are covered in Chapter 9.

Libraries (Worlds and Directories)

Environment libraries constitute the third basic class of Environment objects. Libraries are used for collecting and organizing objects into projects, Ada systems, and the like. There are two kinds of Environment libraries—worlds and directories. At this point, you do not need to know how worlds and directories differ. Between them is of utility when you are ready to systems (see Part V, “Large-Scale Development”). For now, you can think of worlds and directories as equivalent in the following basic respect: Environment worlds and directories contain objects such as files, Ada units, and other worlds or directories.

Environment libraries, therefore, are counterparts to directories on other computer systems you may have used.

Environment worlds and directories are both referred to as *libraries* because both can be used as Ada program libraries as defined by the *Reference Manual for the Ada Programming Language*. That is, for each of your Ada programs, you can create an Environment library that contains the program's compilation units.

As Figure 3-1 shows, a library's name appears underlined in the first line of its display. Also in this line, the library is identified as either a world or a directory; in Figure 3-1, the library !Users.Anderson is identified as a world. Note that home libraries are always worlds, so it is also correct to refer to them as “home worlds.”

The display of a library has an entry representing each object it contains. (The previous two sections showed entries for files and Ada compilation units from the world !Users.Anderson in Figure 3-1.) !Users.Anderson also contains several entries for libraries. These entries indicate whether the library is a world or a directory:

```

Documentation    : Library (Directory);
Tools           : Library (World);

```

Environment Objects and Access Control

Like other computer systems that you may have used, the Environment provides *access control*, which allows you to grant or restrict permission for other users to access libraries, files, or Ada units.

For files and Ada units, either of the following classes of access rights can be granted to particular groups of users:

- *Read access*: Indicates permission to view the object, but not to modify it.
- *Write access*: Indicates permission to both view and modify the object.

For libraries, classes of access rights include permission to view the library display, to create new objects in the library, to change the access classes for objects in the library, and to delete the library.

Default access classes are assigned to each object when it is created; you can change the original access assignment to objects as permitted. For further information, see the Library Management (LM) book of the *Rational Environment Reference Manual*.

The Environment Library Structure

As mentioned in the previous section, libraries can contain other libraries. Such arbitrarily nested libraries form the hierarchic structure in which Environment objects are organized. The Environment's hierarchic library structure is similar to directory structures on other computer systems.

The root of the Environment library structure is a single world called ! (pronounced "bang"). Within the world ! are the basic worlds that support the Environment plus any other worlds or directories that have been created at your installation. The diagram in Figure 3-2 shows a core portion of the Environment library structure, starting with the root world ! and including the sample home library for user Anderson.

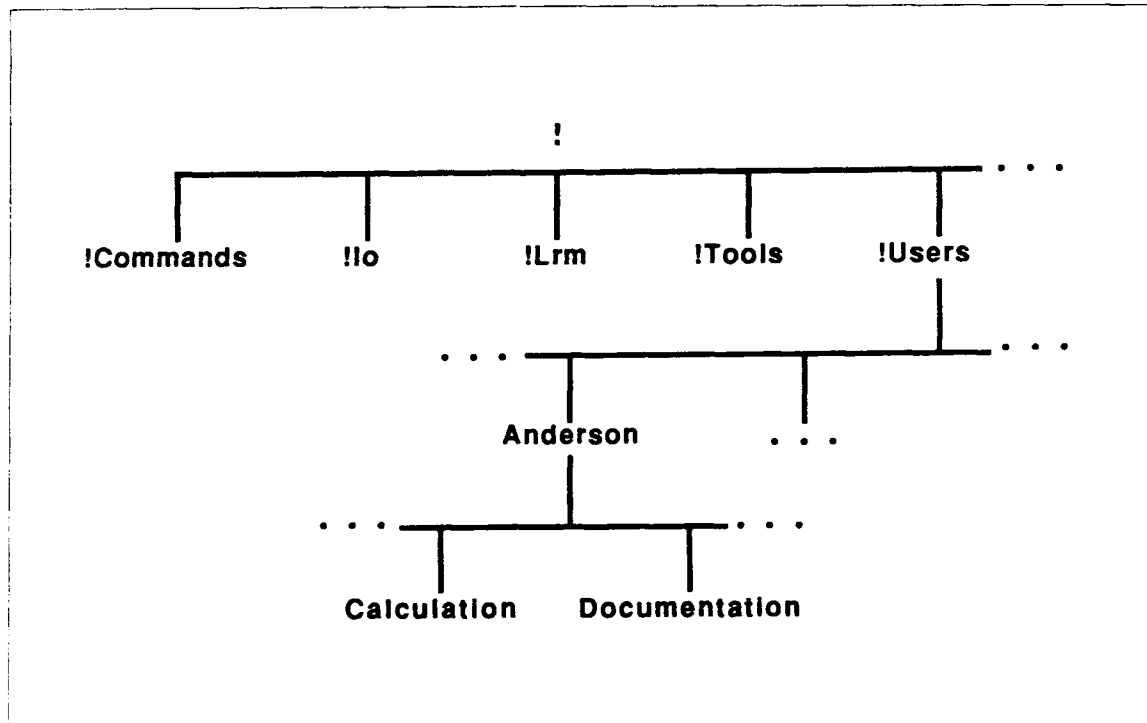


Figure 3-2. The Environment Hierarchy

Figure 3-2 shows five of the worlds in world ! that will be most useful to you. These worlds, which are part of the standard Rational Environment, are described below:

!Commands	Contains the specifications of the Ada packages that define the majority of the Environment commands.
!Io	Contains the specifications of the Ada packages that define the Environment input/output facilities.
!Lrm	Contains specifications of the standard Ada packages (the world's name stands for "Language Reference Manual").
!Tools	Contains predefined Ada tools you can use to build other programs.
!Users	Contains the home libraries for each user in the Environment. For example, as Figure 3-2 shows, the home library for user Anderson is located in the world !Users.

The Reference Summary, in Volume 1 of the *Rational Environment Reference Manual*, lists the complete contents of the worlds !Commands, !Io, !Lrm, and !Tools.

Fully Qualified Object Names

Every object in the Environment has a unique *fully qualified pathname* that reflects the object's unique place in the hierarchic library structure. A fully qualified pathname begins with **!**, which stands for the root world, and contains one or more *name components*, one for each of the libraries in the hierarchy between the root world **!** plus the object itself. The rightmost name component is the object's *simple name*. Multiple name components are separated by periods.

For example, following are the fully qualified names of some of the objects shown in Figure 3-2:

- **!Users** is the name of the world **Users** located within the root world **!**.
- **!Users.Anderson** is the name of the world **Anderson** located within the world **!Users**.
- **!Users.Anderson.Calculation** is the name of the Ada package **Calculation** that is located within the world **!Users.Anderson**.

An Ada unit name such as **!Users.Anderson.Calculation** refers either to the unit's specification or to its body. (Some commands choose one of these as the default reference when you supply this kind of Ada name as a parameter.) To refer specifically to a unit's specification or body, you can append the appropriate *attribute* to the unit's name. An attribute is separated from the unit's simple name by a single quotation mark (**'**):

- **!Users.Anderson.Calculations'Spec** names the specification of **!Users.Anderson.Calculation**.
- **!Users.Anderson.Calculations'Body** names the body of **!Users.Anderson.Calculation**.

The Current Context in the Library Hierarchy

When you view objects in the library hierarchy, those objects are displayed in major windows on the screen. Placing the cursor in one of these major windows helps to establish the *current context*, or "where you are" in the Environment library hierarchy. The current context is an important notion for referring to objects in Environment commands that create objects, view objects, and the like.

The current context is always a library or an Ada unit. Following are guidelines by which you can determine the current context:

- If the object containing the cursor is a library or an Ada unit, the current context is that library or Ada unit.
- If the object containing the cursor is a file, the current context is the library that contains the file.
- For other objects, the current context is determined on a case-by-case basis.
- If the cursor is in a Command window, the current context is determined by contents of the window to which the Command window is attached.

Note that you can make an *absolute* reference to an object from any context by using its fully qualified name. You can also refer to an object *relative* to the current context, using only a portion of the fully qualified name.

For example, you can use the fully qualified name `!Users.Anderson.Calculation` in any context to refer to the package `Calculation` in the world `!Users.Anderson`. In addition, you can refer to this package using a relative name from the following three contexts:

- If the current context is `!Users.Anderson`, you can use the simple name `Calculation`.
- If the current context is `!Users`, you can use the name `Anderson.Calculation`.
- If the current context is `!`, you can use the name `Users.Anderson.Calculation`.

Traversing the Environment Library Structure

When you log in, your home library is displayed automatically by the default login procedure. This section describes how to find and display other objects in the Environment in addition to your home library.

The basic method for finding and displaying objects exploits the hierarchic structure of the Environment. Under this method, you can move, or *traverse*, from object to object within the library hierarchy until you find the object you want. You can traverse downward within the hierarchy by starting in a library and displaying one of the objects that the library contains. You can also traverse upward within the hierarchy, starting at an object and displaying the library that encloses it.

The four basic operations for traversal are bound to modified function key combinations. On the Rational Terminal keyboard overlay, these key combinations are named `Definition`, `Other Part`, `Home Library`, and `Enclosing`. You can find these key combinations in the column labeled “Traverse.”

Besides using the basic traversal operations, you can also cut across the library hierarchy to display any arbitrary object by specifying the object’s name in an Environment command (examples are given in “Ada Usage in Command Windows,” in Chapter 5).

The following sections show how to use Environment commands bound to key combinations to:

- Traverse from a library to an object in that library (`Definition`)
- Traverse between an Ada specification and the corresponding body in either direction (`Other Part`)
- Traverse from an object to the library that encloses it (`Enclosing`)
- Return to your home library from any context in the Environment (`Home Library`)

Traversing from a Library to an Object in It

When a library like your home library is displayed in a window, you can use its display to bring up a window for any of the objects in that library. This way of viewing, or *getting the definition of*, objects involves using `Definition`.

For example, assume that you are user Anderson and you want to view the contents of one of the libraries listed in your home library—for example, the Tools world.

Starting with the screen as it appears at login (recall Figure 3-1):

1. Move the cursor into the window that contains your home library (see “Moving between and within Windows,” above.)
2. Use the cursor keys to position the cursor on the entry for the Tools world. You can put the cursor anywhere on the line containing the entry. Note that in long libraries, you may need to scroll through the library entries to find the desired one (see “Moving between and within Windows,” above).

Figure 3-3 shows your home library with the cursor next to the entry for the Tools world:

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Rational: {Delta} ANDERSON S_1
-----
!Users Anderson : Library (World):
Calculation    : I Ada (Pack_Spec);
Calculation    : I Ada (Pack_Body);
Documentation  : Library (Directory);
Factorial      : S Ada (Func_Spec);
Factorial      : S Ada (Func_Body);
Login          : C Ada (Proc_Spec);
Login          : C Ada (Proc_Body);
Memo_12_08_86 : File (Text);
My_Test_Data  : File (Binary);
S_1           : Session;
S_1_Switches  : File (Switch);
Tools         : Library (World);
= !USERS ANDERSON: (library) World
  {statement}
end;
```

Figure 3-3. Designating the Tools World for Viewing

3. Press `Definition`.

`Definition` opens another major window and displays the designated object, `!Users-Anderson.Tools`, in it. The cursor is placed in the new window at the beginning of the first line, as shown in Figure 3-4:

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Rational: (Delta) ANDERSON S_1
-----
!Users_Anderson : Library (World):
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Documentation     : Library (Directory);
Factorial        : S Ada (Func_Spec);
Factorial        : S Ada (Func_Body);
Login            : C Ada (Proc_Spec);
Login            : C Ada (Proc_Body);
Memo_12_08_86   : File (Text);
My_Test_Data    : File (Binary);
S_1             : Session;
S_1_Switches    : File (Switch);
Tools           : Library (World);

= !USERS-ANDERSON-(library) : World
[PROCEDURE]
end.

!Users_Anderson.Tools : Library (World):
Math_Tools       : C Ada (Pack_Spec);
Math_Tools       : C Ada (Pack_Body);
Scan_Tools       : C Ada (Pack_Spec);
Scan_Tools       : C Ada (Pack_Body);
String_Tools     : C Ada (Pack_Spec);
String_Tools     : C Ada (Pack_Body);

= !USERS-ANDERSON-TOOLS : (library) : World
```

Figure 3-4. After Pressing `Definition`

You can use `Definition` to view any kind of objects, including Ada units and files. When you get the definition of an Ada unit or a file, the display is read-only; to make changes to an Ada unit or a file, you must make an explicit request to open it for editing (see Chapters 7 and 10).

For example, now that you have displayed `!Users.Anderson.Tools`, you can use `Definition` to display one of the units in this world—say, the specification for package `String_Tools`:

1. In the world `!Users.Anderson.Tools`, move the cursor to the entry for the `String_Tools` package specification, as shown in Figure 3-5:

```
Rational Environment
  D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Rational:Delta: ANDERSON.S_1
-----
!Users.Anderson : Library (World):
Calculation    : I Ada (Pack_Spec);
Calculation    : I Ada (Pack_Body);
Documentation  : Library (Directory);
Factorial      : S Ada (Func_Spec);
Factorial      : S Ada (Func_Body);
Login          : C Ada (Proc_Spec);
Login          : C Ada (Proc_Body);
Memo_12_08_86  : File (Text);
My_Test_Data   : File (Binary);
S_1            : Session;
S_1_Switches  : File (Switch);
Tools         : Library (World);

= !USERS:ANDERSON (library) world
[statement]
end;

-----
!Users.Anderson.Tools : Library (World):
Math_Tools     : C Ada (Pack_Spec);
Math_Tools     : C Ada (Pack_Body);
Scan_Tools     : C Ada (Pack_Spec);
Scan_Tools     : C Ada (Pack_Body);
String_Tools   : C Ada (Pack_Spec);
String_Tools   : C Ada (Pack_Body);

= !USERS:ANDERSON:TOOLS (library) world
```

Figure 3-5. Designating the Specification for Package `String_Tools`

2. Press **Definition**. The designated Ada unit is displayed with read-only access in a third major window, as shown in Figure 3-6:

```

Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
Rational (Delta) ANDERSON $_1
-----
!Users Anderson : Library (World):
Calculation : I Ada (Pack_Spec);
Calculation : I Ada (Pack_Body);
Documentation : Library (Directory);
Factorial : S Ada (Func_Spec);
Factorial : S Ada (Func_Body);
Login : C Ada (Proc_Spec);
Login : C Ada (Proc_Body);
Memo_12_08_86 : File (Text);
My_Test_Data : File (Binary);
S_1 : Session;
S_1_Switches : File (Switch);
Tools : Library (World);
-----
-!USERS ANDERSON (library) ~ World
[Statement]
end;
-----
!Users Anderson Tools : Library (World):
Math_Tools : C Ada (Pack_Spec);
Math_Tools : C Ada (Pack_Body);
Scan_Tools : C Ada (Pack_Spec);
Scan_Tools : C Ada (Pack_Body);
String_Tools : C Ada (Pack_Spec);
String_Tools : C Ada (Pack_Body);
-----
-!USERS ANDERSON TOOLS (library) World
-----
package String_Tools is
function Equal (String1 : String; String2 : String;
Ignore_Case : Boolean := True) return Boolean;
function Greater_Than (String1 : String; String2 : String;
Ignore_Case : Boolean := True) return Boolean;
function Less_Than (String1 : String; String2 : String;
Ignore_Case : Boolean := True) return Boolean;
function Lower_Case (S : String) return String;
function Upper_Case (S : String) return String;
end String_Tools;
-----
-!USERS ANDERSON TOOLS $STRING_TOOLS V(5) (ada) Coded

```

Figure 3-6. After Pressing **Definition**

The use of `Definition` illustrates a basic pattern underlying Environment operations. This pattern can be characterized as a “noun-verb” pattern. According to this pattern, two-part operations have the following order:

1. An object is designated—for example, by cursor position.
2. An action is indicated—for example, by pressing `Definition`.

This pattern is consistent with the item-operation type of key combination (see Chapter 1).

Traversing between Ada Specifications and Bodies

Having displayed the specification for the Ada package `String_Tools`, you can easily bring up a window for the body of this package, as well. There are several ways to do this. You can always use the methods described in the previous section—namely, moving the cursor back into the Tools library, putting the cursor on the entry for the `String_Tools` package body, and then pressing `Definition`. Alternatively, you can traverse directly to the package body from its specification by pressing `Other Part`.

`Other Part` uses the Environment’s underlying Ada unit representation to find an Ada unit’s body from its specification or to find an Ada unit’s specification from its body. You can use `Other Part` with Ada procedures, functions, packages, and generics.

For example, to traverse from the specification of package `String_Tools` to its body:

1. Leave the cursor in the window containing the package specification, as in Figure 3-6 on the previous page. The cursor can be anywhere in this window.
2. Press `Other Part`.

On the next page, Figure 3-7 shows that `Other Part` opens a new major window and displays `!Users.Anderson.Tools.String_Tools’Body` in it. Like `Definition`, `Other Part` displays Ada units with read-only access.

Note that, in previous examples, new windows were simply added below existing ones. In this example, however, the new window has replaced the window containing your home library, so that the new window appears at the top of the screen. This is because, by default, the screen can contain only three major windows at a time (you can change this default; see Chapter 4, “Managing Windows”). Therefore, when you display a fourth object, its window must replace a window that is already on the screen. Chapter 4 describes the order in which windows are replaced.

Rational Environment

D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.

= Rational (Delta) ANDERSON \$_1

```

package body String_Tools is
  function Equal (String1 : String; String2 : String;
                 Ignore_Case : Boolean := True) return Boolean is
  begin
    if Ignore_Case then
      declare
        Upper1 : constant String := Upper_Case (String1);
        Upper2 : constant String := Upper_Case (String2);
      begin
        return Upper1 = Upper2;
      end;
    else
      return String1 = String2;
    end if;
  end Equal;
end String_Tools;

```

= .TOOLS.STRING_TOOLS'BODY'V16' (ada) Coded**!Users.Anderson.Tools : Library (World):**

```

Math_Tools : C Ada (Pack_Spec);
Math_Tools : C Ada (Pack_Body);
Scan_Tools : C Ada (Pack_Spec);
Scan_Tools : C Ada (Pack_Body);
String_Tools : C Ada (Pack_Spec);
String_Tools : C Ada (Pack_Body);

```

= !USERS-ANDERSON-TOOLS- (library) ~ World

```

package String_Tools is
  function Equal (String1 : String; String2 : String;
                 Ignore_Case : Boolean := True) return Boolean;
  function Greater_Than (String1 : String; String2 : String;
                        Ignore_Case : Boolean := True) return Boolean;
  function Less_Than (String1 : String; String2 : String;
                     Ignore_Case : Boolean := True) return Boolean;
  function Lower_Case (S : String) return String;
  function Upper_Case (S : String) return String;
end String_Tools;

```

= !USERS-ANDERSON-TOOLS-STRING_TOOLS'V16' (ada) CodedFigure 3-7. After Pressing Other Part

Returning Home

You can use `Home Library` to return to your home library directly from any context in the Environment library structure. For example, now that you have traversed to another library and viewed several objects in that library, you can bring your home library back onto the screen as a starting point for further traversal.

To return home:

Press `Home Library`. The cursor can be in any window on the screen.

As shown in Figure 3-8 on the following page, your home library is redisplayed, replacing another window as necessary. The cursor is returned to the entry for the Tools world, which is the cursor's most recent position in that window.

Using `Home Library` is a convenient way of retrieving your home library once it has been replaced. (Note that Command windows, such as the one attached to your home library, are replaced and redisplayed along with the major windows to which they are attached.) If you press `Home Library` while the home library is still displayed, the cursor simply moves into the appropriate window.

```

Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Rational | Delta | ANDERSON S_1
-----
package body String_Tools is
  function Equal (String1 : String; String2 : String;
                 Ignore_Case : Boolean := True) return Boolean is
  begin
    if Ignore_Case then
      declare
        Upper1 : constant String := Upper_Case (String1);
        Upper2 : constant String := Upper_Case (String2);
      begin
        return Upper1 = Upper2;
      end;
    else
      return String1 = String2;
    end if;
  end Equal;
= TOOLS.STRING_TOOLS'BODY'V161 (ada) Coded
-----
!Users Anderson : Library (World):
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Documentation     : Library (Directory);
Factorial         : S Ada (Func_Spec);
Factorial         : S Ada (Func_Body);
Login             : C Ada (Proc_Spec);
Login             : C Ada (Proc_Body);
Memo_12_08_86    : File (Text);
My_Test_Data     : File (Binary);
S_1               : Session;
S_1_Switches     : File (Switch);
! Tools          : Library (World);
= ANDERSON-ANDERSON-!libraryn World-
! Documents
end;
-----
package String_Tools is
  function Equal (String1 : String; String2 : String;
                 Ignore_Case : Boolean := True) return Boolean;
  function Greater_Than (String1 : String; String2 : String;
                        Ignore_Case : Boolean := True) return Boolean;
  function Less_Than (String1 : String; String2 : String;
                     Ignore_Case : Boolean := True) return Boolean;
  function Lower_Case (S : String) return String;
  function Upper_Case (S : String) return String;
end String_Tools;
= !USERS ANDERSON-TOOLS.STRING_TOOLS'V161 (ada) ~ Coded
-----

```

Figure 3-8. After Pressing

Traversing to the Enclosing Library

In contrast to `Definition`, which traverses “down” the Environment hierarchy, `Enclosing` enables you to traverse “up” the Environment hierarchy. That is, using `Enclosing`, you can start from an object and traverse to the library that contains, or encloses, that object.

For example, to view the world `!Users`, you can use `Enclosing` from your home library, since the world `!Users` encloses your home library.

To display the world `!Users`:

1. Start with the cursor in your home library, as in Figure 3-8.
2. Press `Enclosing`. Use the keyboard overlay to locate `Enclosing` among the function keys. Note that `Enclosing` is in the column labeled “Traverse” on the keyboard overlay.

As shown in Figure 3-9 on the following page, the world `!Users` is displayed, replacing another window as necessary. The cursor appears next to the entry for Anderson and this entry is highlighted, because you traversed from the world `!Users.Anderson`.

Note that, with the cursor in the window for `!Users`, you have several options for further traversal:

- You can press `Enclosing` one more time to view the root world `!`. Note that you cannot use `Enclosing` to traverse beyond the world `!` because no further world encloses it.
- You can reposition the cursor within the world `!Users` and then use `Definition` to display another user’s home library. For example, if you need to read a memo in user Miyata’s home library, you can move the cursor to the entry for Miyata and press `Definition` to display `!Users.Miyata`. Then you can use `Definition` again to display the desired memo.

Rational Environment

D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.

⊞ Rational ⊞ Delta ANDERSON S_1

```

package body String_Tools is
  function Equal (String1 : String; String2 : String;
                 Ignore_Case : Boolean := True) return Boolean is
  begin
    if Ignore_Case then
      declare
        Upper1 : constant String := Upper_Case (String1);
        Upper2 : constant String := Upper_Case (String2);
      begin
        return Upper1 = Upper2;
      end;
    else
      return String1 = String2;
    end if;
  end;

```

⊞ Tools ⊞ STRING_TOOLS ⊞ BODY ⊞ V.6 ⊞ Ada ⊞ ~ Coded

```

!Users Anderson : Library (World);
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Documentation     : Library (Directory);
Factorial        : S Ada (Func_Spec);
Factorial        : S Ada (Func_Body);
Login            : C Ada (Proc_Spec);
Login            : C Ada (Proc_Body);
Memo_12_08_86   : File (Text);
My_Test_Data    : File (Binary);
S_1              : Session;
S_1_Switches    : File (Switch);
Tools            : Library (World);

```

⊞ USERS ⊞ ANDERSON ⊞ library ⊞ World**⊞ Statements**

end;

!Users : Library (World):

```

Anderson : Library (World);
Bate     : Library (World);
Bes      : Library (World);
Blb      : Library (World);
Bls      : Library (World);
Bolz     : Library (World);
Dbh      : Library (World);
Dbp      : Library (World);
Dce      : Library (World);
Dcg      : Library (World);
Demo     : Library (World);
Demo10   : Library (World);
Demo11   : Library (World);

```

⊞ USERS ⊞ library ⊞ WorldFigure 3-9. Displaying !Users with **⊞ Enclosing**

Traversing the Environment: Summary

The examples in the previous four sections showed how to traverse a portion of the Environment library structure using `Definition`, `Other Part`, `Home Library`, and `Enclosing`. The diagram in Figure 3-10 shows the objects that were viewed in the previous examples, indicating the sequence of keys that was used:

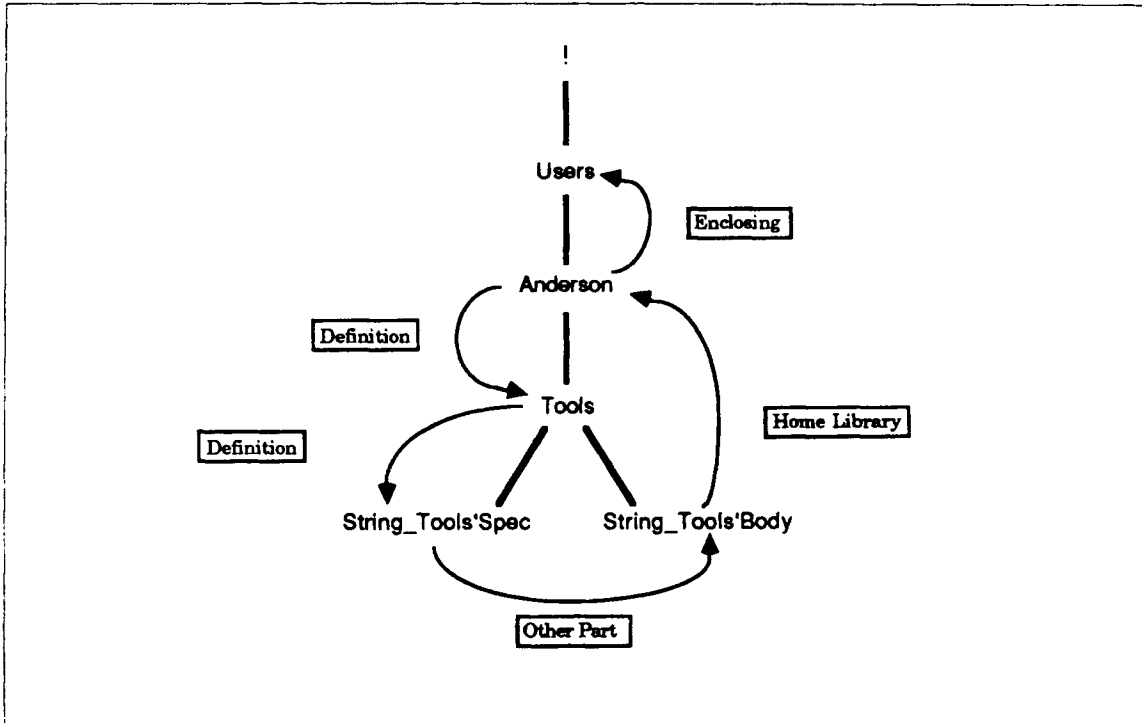


Figure 3-10. The Objects Viewed Using Traversal Operations

Figure 3-10 represents only one of several possible ways to traverse these objects, because you can always use combinations of `Definition` and `Enclosing` as less direct equivalents of `Other Part` and `Home Library`.

For example, instead of using `Other Part` to traverse from the specification of package `String_Tools` to the body, you can use `Enclosing` to return to the `Tools` world, and then use `Definition` to display `String_Tools'Body`. Both alternatives are represented in Figure 3-11, one by a solid line and one by a dotted line:

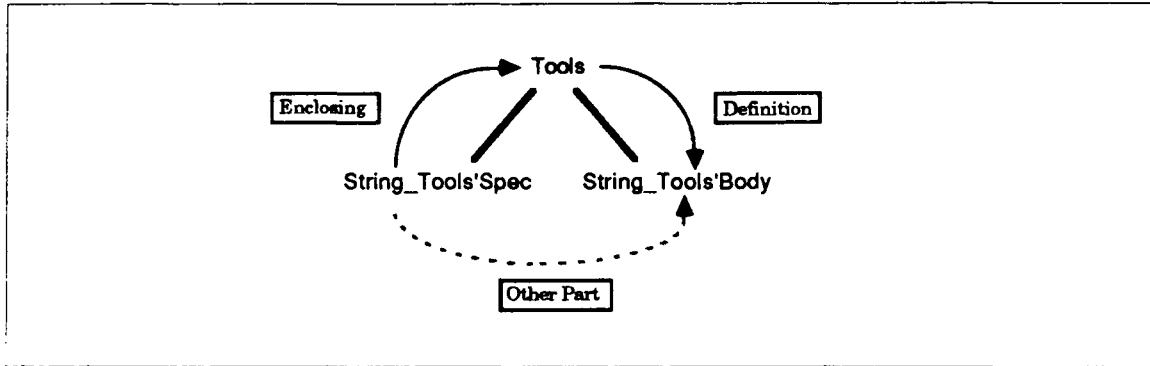


Figure 3-11. An Alternate Route from `String_Tools` Specification to `Body`

Similarly, instead of using `Home Library` to traverse from `String_Tools'Body` to your home library, you can “retrace your steps” by pressing `Enclosing` twice, as shown in Figure 3-12:

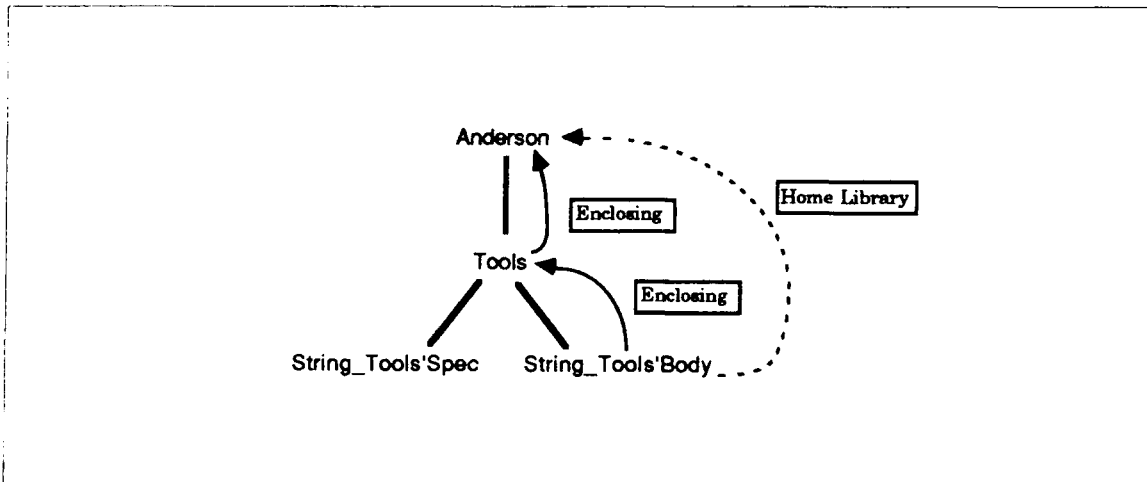


Figure 3-12. An Alternate Route Home from `String_Tools'Body`

When you traverse to an object, that object is displayed in a window on the screen and the cursor is placed in the object's window. As shown in the previous set of examples, windows are added to the screen until the screen is full, after which existing windows are replaced. (See Chapter 4 for further information about windows.) Note that if you traverse to an object whose window is still on the screen, the cursor simply moves to that window. (In this case, you can equivalently use `Window` - `↑` or `Window` - `↓` to move the cursor.)

As you traverse the Environment, bear in mind that Environment objects are subject to access control. In particular, if you try to traverse to an object to which read access has been restricted, a message is displayed indicating that you cannot view that object. (For more information on access control, see the Library Management (LM) book of the *Rational Environment Reference Manual*.)

Finally, it is important to note that the traversal operations `Definition`, `Enclosing`, and `Other Part` can be used to browse Ada systems (see Chapter 14). You can think of traversing the library structure as a special case of this more general usage.

Chapter 4. Managing Windows

When you log in, your screen contains several windows. As you display more objects, more windows are created. This chapter describes the various ways in which you and the Environment can manage multiple windows. Specifically, this chapter describes:

- How objects and information are displayed in windows
- How multiple windows are placed on the screen
- How you can redisplay windows that have been replaced
- How you can change the number, size, and location of the windows on the screen

Windows and Images

Work in the Environment is accomplished through various kinds of windows. Three kinds of windows were introduced in “What You See When You Log In,” in Chapter 3—namely, the Message window, major windows, and Command windows. A fourth kind of Environment window, called *minor windows*, is introduced in Chapter 15. Each kind of Environment window contains, or displays, objects and other information. More precisely, Environment windows contain the displayable representations, or *images*, of objects and information.

An image is a user-readable textual representation that is suitable for displaying in a window. Sometimes it is useful to refer specifically to images—for example, when describing operations such as scrolling, which act exclusively on displayed images. Typically, however, reference to an image is omitted where it can be inferred. That is, in this guide, you will see phrases like “the window containing the object” rather than “the window containing the image of the object.”

Scrolling an Image

At any one time, a window actually displays only a portion of the entire image of an object. This is because each Environment window is finite in size, whereas an object’s image can extend arbitrarily far to the right or down. You can think of an image as lines of characters surrounded by an indefinite expanse of white space.

The diagrams in Figure 4-1 represent various ways in which you can position a window relative to the image it contains. In diagram A, the top of the image (the first line) coincides with the top of the window, and the left edge of the image

coincides with the window's left edge. Most images are displayed this way initially; you can always bring the top of an image into view using `Image - Begin Of`.

Diagram A shows that the lines toward the bottom of the image are not visible within the window. You can bring these lines into view by using `Image - ↓` to scroll down by one full window of lines, as indicated in diagram B. Conversely, `Image - ↑` scrolls up by one full window of lines.

Diagram A also shows that an image can extend beyond the right edge of the window. You can bring this portion of the image into view by using `Image - ←`. Each use of this key combination scrolls 16 columns to the right (diagram C). Conversely, `Image - →` scrolls 16 columns to the left.

Finally, you can use `Image - End Of` to display the last full window of lines directly, without viewing any intervening lines (diagram D).

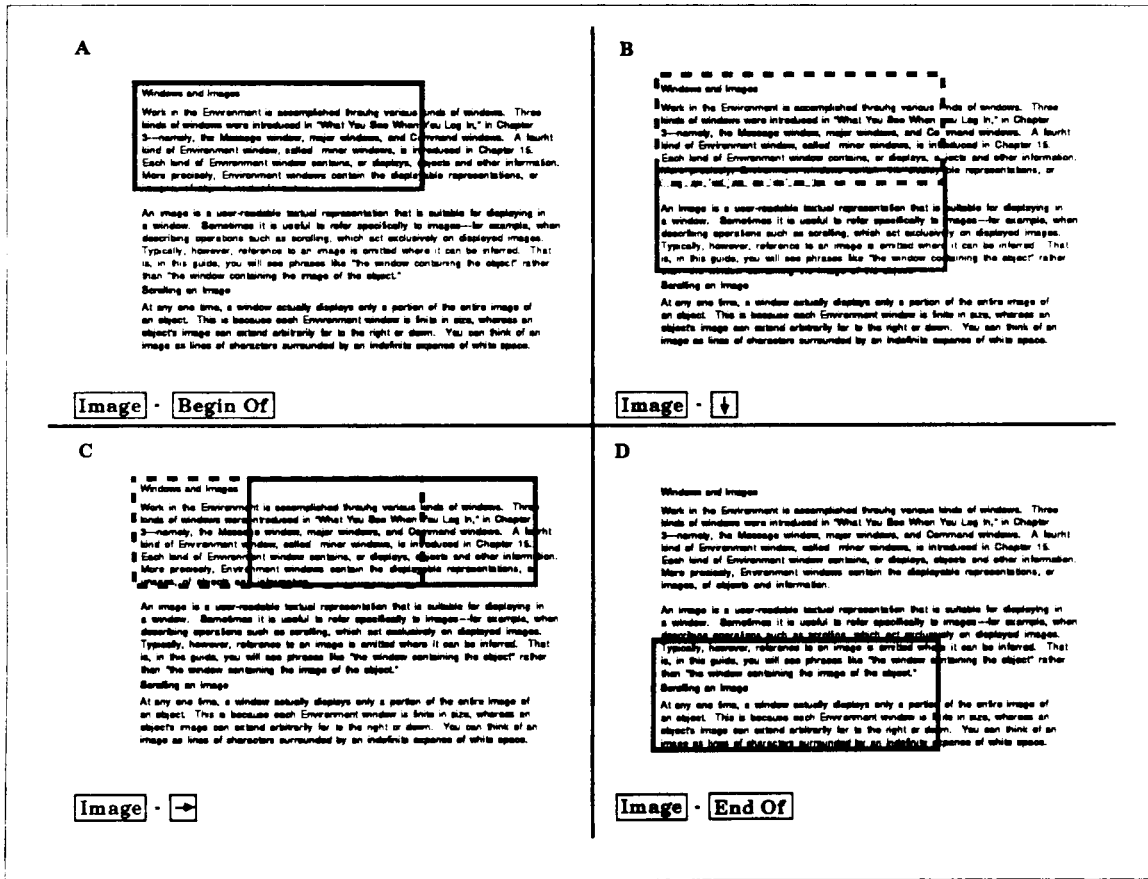


Figure 4-1. Displaying Different Portions of an Image

Window Banners

All windows except Command windows display a banner beneath the visible portion of the image. The banner is a reverse video line containing information about the image in the window. In the banners for major windows, this information falls into several fields, as shown in Figure 4-2:

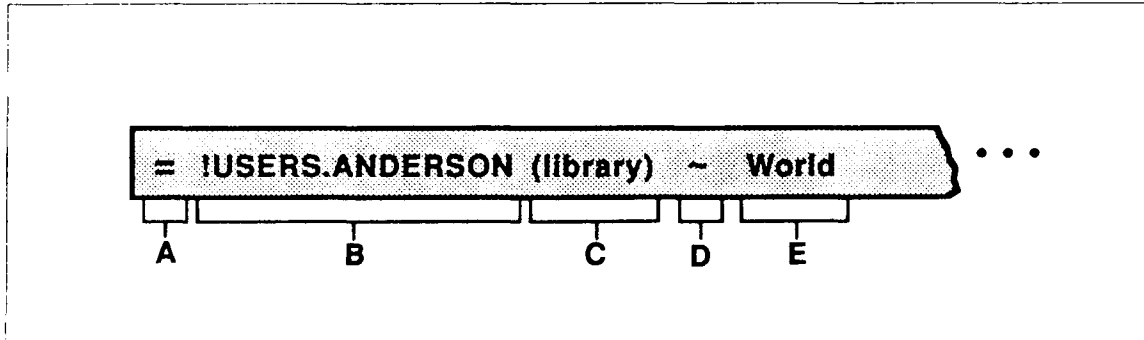


Figure 4-2. The Fields of a Typical Window Banner

- A. The leftmost field of the banner contains a symbol that indicates whether the image can be modified and, if so, whether it actually has been modified. These symbols are described in “Modification Symbols in the Window Banner,” below.
- B. The next field names the contents of the window. A fully qualified name appears for objects like Ada units, files, and libraries. When a job opens a window for input or output, this field in the banner displays the name of the context from which the job was initiated, followed by the job name. Long names are truncated on the left; truncation is indicated by three dots preceding the name fragment.
- C. Following the name and enclosed in parentheses is an indication of the editing facilities that are available for the object in the window. This typically corresponds to the class or subclass of the displayed object. Typically, you will see `(ada)`, `(text)`, or `(library)` here.
- D. The next field may contain a blank, a tilde (`~`), or an *at* sign (`@`). The tilde indicates the next window to be replaced, and the *at* sign indicates a window that cannot be replaced; see “How Windows Are Placed on the Screen,” below.
- E. The rightmost field of the banner contains further information about the contents of the window. For Ada units, the unit state (for example, *Source*, *Installed*, or *Coded*) is displayed (see Chapter 9). For libraries, the subclass (*World* or *Directory*) is displayed.

The banner for the Message window is unique. This banner contains the name of your installation’s R1000, your username, and the name of the session you are logged into. Furthermore, this banner is where certain messages, such as the `...running` message, are displayed.

Modification Symbols in the Window Banner

When you edit an object such as an Ada unit or a file, the changes you make appear in the image, where you can view them. However, the changes you make to an image are not permanent. To update the object itself, you must explicitly save, or *commit*, the changes.

The symbols in the leftmost field of a window banner indicate whether the image in a window can be modified. For modifiable images, these symbols indicate whether the image has in fact been modified. The modification symbols include:

- = Indicates that the image is read-only. Some images are inherently read-only—for example, images of libraries. Images of files and Ada units are read-only if you have not explicitly opened such objects for editing. For example, using `Definition` displays a file or an Ada unit with a read-only image.
- (blank) Indicates that the image is modifiable and that no changes have been made since the last time it was saved. The blank symbol appears in the banner of an unchanged file or Ada unit that is open for editing.
- * Indicates that the image is modifiable and that changes have been made to it since the last time it was saved. In a window created by a job, the * symbol indicates that the job requires interactive input.
- # Indicates that the image of an Ada unit is modifiable and that the image has been changed since the last time it was saved; additionally, these changes have been assimilated into an internal representation through an operation called *formatting* (see Chapter 11, “Using Ada-Specific Editing Operations”). In a job window, the # indicates that the job is requesting interactive input.
- ! Indicates that the image is currently read-only because a job has obtained access to the object in that window.

Note that the Quit command will fail if a window banner contains an * or # symbol when you log out.

How Windows Are Placed on the Screen

As you traverse the Environment to view and edit objects, windows are automatically opened on the screen. When the available screen space is full, new windows cause existing windows to be replaced. The following sections describe the Environment's basic strategy for placing windows on the screen and the various ways you can further control the placement of windows.

Windows and Frames

The Environment divides the screen space below the Message window into *frames* for the purpose of managing window placement. Each frame is a separate logical area on the screen in which a major window can be placed. Any windows that are attached to a major window—namely, Command windows and minor windows—share the same frame along with the major window to which they are attached. Frames are not visually represented on the screen; they are marked only by the boundaries of the windows within them.

The diagram in Figure 4-3 represents a screen that is divided into three frames, whose boundaries are indicated with dotted lines. Each frame contains a major window; the top and bottom frames contain major windows with attached Command or minor windows. Note that the Message window itself is not in a frame and is fixed at the top of the screen.

Frames define the basic number, size, and screen location of major windows. By default, the Rational Terminal screen is divided into three frames (terminals with smaller screens default to fewer frames). Since each frame can contain at most one major window, the default Rational Terminal screen can contain up to three major windows at a time. You can increase the number of frames to view more objects; conversely, you can decrease the number of frames to view fewer objects in larger windows. (See "Changing the Number of Frames," below.)

Also by default, frames divide the screen into equal portions, so that changing the number of frames also changes the basic size for each individual frame. However, the basic frame size is flexible—that is, frames can stretch or shrink when you resize the windows in them (see "Changing Window Size and Placement," below). Enlarging windows is especially useful when several attached windows share a single frame.

Note that frames can be empty, as they are before windows have been placed in them or after windows have been deleted. Empty frames appear as empty screen space.

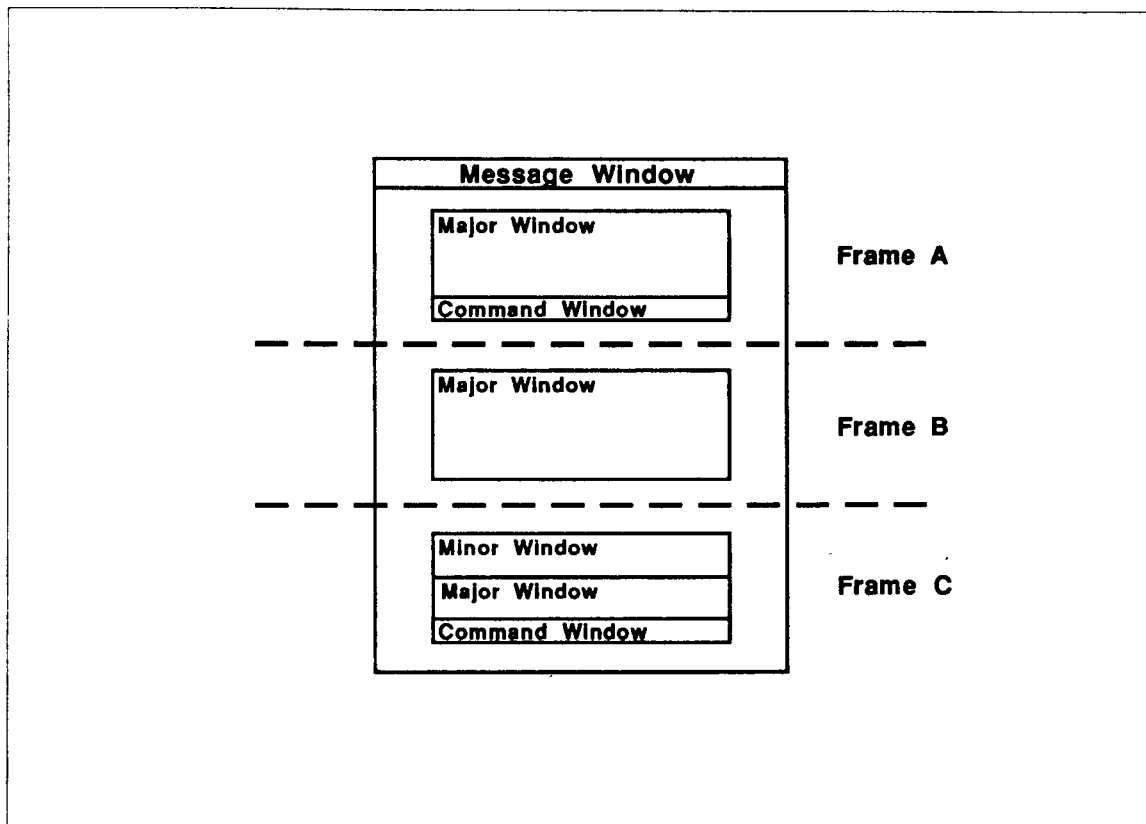


Figure 4-3. A Screen with Windows in Three Frames

Moving between Windows

The key combinations `Window - ↑` and `Window - ↓` move the cursor between windows, either within or between frames.

For example, assume that the cursor is in the major window in the top frame in Figure 4-3 above. Pressing `Window - ↓` moves the cursor into the attached Command window within the same frame. Pressing `Window - ↓` again moves the cursor into the major window in the middle frame. (Note that pressing `numeric 9 - Window - ↓` is the same as pressing `Window - ↓` twice.)

When you move the cursor far enough in either direction, the cursor wraps around the screen. For example, when the cursor is in the bottommost window on the screen, pressing `Window - ↓` puts the cursor in the Message window at the top of the screen.

Default Window Placement

The Environment automatically opens a major window for each object you view or edit and for each job that produces a display or requires interactive input. By default, a major window is placed on the screen as follows:

- If there is an empty frame on the screen, the window is placed in that frame.
- If all frames are full, the window is placed in the *least recently visited* frame, replacing the major window plus any attached windows that were in that frame.

To *visit* a frame is to move the cursor into a window in that frame—for example, by using `Window - F` or `Window - I` or by using traversal operations such as `Definition`. If a frame contains more than one window (say, a major window with an attached Command window), putting the cursor into any one of those windows counts as visiting the frame. The least recently visited frame is the frame whose windows have gone the longest without being touched by the cursor. The Environment assumes that the unvisited frames contain the windows of least current interest, so these windows are the best candidates for replacement by a more recently requested object.

The major window in the least recently visited frame displays a tilde (~) in its banner, so that you do not have to keep track of the order in which you visited each frame. Figure 4-2 shows where the tilde appears in a window banner. This visual indicator simplifies what you have to remember about window placement—namely, the next major window to be replaced displays a tilde (~) in its banner.

Controlling Window Placement by Locking Windows

You can lock a major window on the screen to prevent it from being replaced by a more recently requested window. Once locked, a window is no longer subject to the “least recently visited” condition, so that it remains on the screen regardless of whether the cursor has visited its frame. For example, if you are working on an Ada unit specification and body and you periodically need to look at other related units, you can lock the two windows you are working on so that subsequent traversal will not cause them to be replaced.

To lock a major window on the screen:

1. Make sure the cursor is in the window you want to lock.
2. Press `Window - Promote`.
3. Note that an *at* sign (@) is displayed in the banner of the locked window. The *at* sign appears in the same position in the window banner as the tilde (see Figure 4-2, above).

A locked window remains locked until you explicitly unlock it. When you unlock a window, it becomes eligible for replacement again.

To unlock a locked major window:

1. Make sure the cursor is in the window you want to unlock.
2. Press `Window` - `Demote`. You can find `Demote` on the keyboard overlay in the column labeled "Demote."
3. Note that the *at* sign (@) is removed from the banner of the window you just unlocked.

You can force an unlocked window to become the next window to be replaced, independently of the "least recently visited" condition. Designating a window for replacement is a quick way to make sure that other windows remain on the screen without locking them, at least for the next traversal operation.

To designate a window for replacement:

1. Make sure the cursor is in the appropriate window.
2. Press `Window` - `Demote`.
3. Note that the tilde (~) appears in the banner of the window, indicating that this window will be replaced next. The tilde remains in this window even if you visit it with the cursor.

Controlling Window Placement through Traversal Commands

The traversal operations you have seen so far display objects using default window placement. That is, when you display an object using `Definition`, `Other Part`, `Enclosing`, or `Home Library`, a major window is opened in the least recently visited frame.

Two variant traversal operations offer an alternative to default window placement. These are `Definition In Place` and `Enclosing In Place`. Both of these keys are identical to their counterparts (`Definition` and `Enclosing`, respectively), except that the new keys display objects in place, as their names suggest. That is, when you use either `Definition In Place` or `Enclosing In Place`, the window for the requested object is placed in the current frame (the frame that contains the cursor) rather than in the least recently visited frame.

Note that the presence or absence of the tilde in window banners is irrelevant to both `Definition In Place` and `Enclosing In Place`. Furthermore, both keys cause the current window to be replaced even if that window is locked.

Redisplaying Windows Using the Window Directory

The Environment maintains a list of the major windows that have been opened since you logged in. This list, called the *Window Directory*, has an entry for each object you have displayed, for each job involving input or output, and for Environment services such as the Message window and the Window Directory itself.

When a window is replaced by a more recently requested window, the replaced window retains its entry in the Window Directory. Window Directory entries provide an easy way to redisplay replaced windows. In addition, Window Directory entries indicate whether or not images in windows have been modified, even for images in replaced windows that are currently not on the screen.

Displaying the Window Directory

To display the Window Directory:

Press `Window` - `Definition`. The cursor can be in any window on the screen.

The Window Directory is displayed in a major window in the least recently visited frame.

A sample Window Directory is shown in Figure 4-4. Note that the banner of this window identifies it as the Window Directory. The tilde (~) in the banner indicates that the Window Directory will be the next window to be replaced, regardless of when other windows on the screen were visited.

```

Rational Environment
  D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
  ~ Rational (Delta) ANDERSON 9_1
=====
Mod  Lines      Type      Buffer Name
===  =====
*    1  (text)    !USERS.ANDERSON.MEMO_12_08_86'V(2)
      22 (text)    !USERS.ANDERSON.%WHAT.USERS
#    73 (ada)    !USERS.ANDERSON.TOOLS.STRING_TOOLS'BODY'V(6)
=    11 (ada)    !USERS.ANDERSON.TOOLS.STRING_TOOLS'V(6)
=     9 (library) !USERS.ANDERSON.TOOLS
=    14 (library) !USERS.ANDERSON
      1 Help Window
=    29 Message Window
=    11 (windows) Window Directory
=====

```

Figure 4-4. The Window Directory

Information about each Window Directory entry is arranged in four columns. From left to right, the Window Directory entry for a given window shows the following:

Mod	Indicates whether the image in the window can be or has been modified. This column contains the modification symbol that appears in the leftmost field of that window's banner (see "Modification Symbols in the Window Banner," above). Using this column, you can tell at a glance whether any images contain unsaved changes—namely, those with * or # in the Mod column.
Lines	Indicates the number of lines of text displayed in the window.
Type	Indicates the editing facilities that are available for the object in the window. This indication appears enclosed in parentheses in that window's banner and typically corresponds to the class or subclass of the displayed object.
Buffer Name	Indicates what is displayed in the window. The name that appears here is the same as the name that appears in that window's banner. Note that long names are truncated on the left; truncation is indicated by three dots preceding the name fragment.

Redisplaying Replaced Windows

You can use the Window Directory to bring replaced windows back to the screen. Using the Window Directory can be a convenient alternative to redisplaying objects using traversal operations.

To redisplay a window using the Window Directory:

1. Display the Window Directory (see the previous section). Leave the cursor in the Window Directory window.
2. Check the Buffer Name column in the Window Directory until you find the entry for the window you want to redisplay.
3. Place the cursor anywhere on the line containing the appropriate entry.
4. Press `Definition`.

The window you requested is displayed in an empty frame or else it replaces the Window Directory window. Note that the redisplayed window is brought back to the screen along with any attached Command windows or minor windows.

Note that you have seen two uses for `Definition` so far. You can use `Definition` in:

- Libraries to display objects listed there
- The Window Directory to redisplay windows listed there

Further uses for `Definition` are covered in Chapter 14, "Browsing Ada Programs."

Checking the Window Directory Before Logging Out

You can use the Window Directory for other operations besides redisplaying windows. For example, you can display the Window Directory before logging out to see whether or not any images contain unsaved changes. (If there are modified images, the Quit command will fail; see “Logging Out,” in Chapter 2.) You can save the changes in all images directly from the Window Directory. Saving changes from the Window Directory is equivalent to redisplaying individual windows and performing the operation from there.

More specifically, say you have entered the Quit command and the following message is displayed in the Message window:

There are uncommitted images.

You can take the following steps:

1. Display the Window Directory.
2. Check the Mod column in the Window Directory. Entries for modified images have * or # in this column.
3. At this point, you have several alternatives:
 - You can use `Definition` to redisplay each window individually so that you can inspect the changes. To save the changes in a given window, press `Enter`.
 - You can save all changes in all images directly from the Window Directory by pressing `Enter`. The cursor can be on any line in the Window Directory window.
 - You can log out and discard any unsaved changes by entering the Quit command with the parameter True. (See “Logging Out with Unsaved Changes,” in Chapter 2.)

The Editing Specific Types (EST) book of the *Rational Environment Reference Manual* contains information about other possible operations, such as saving changes in selected images from the Window Directory.

Changing Window Size and Placement

Once on the screen, windows can be resized, removed, and rearranged. You can:

- Expand or shrink any window to view more or less of the object it contains.
- Delete windows when you no longer want to view them.
- Change the top-to-bottom order of the windows on the screen—for example, to bring two objects closer together for easy comparison or to bring a particular object to eye level.

Joining Frames

When you need fewer and larger windows (for example, a single full-length window in which to view an entire page of documentation), you can join the frames pairwise to adjacent frames. When two frames are joined, they form a single, larger frame that occupies the screen space of the two original frames. The contents of one of the original frames is displayed in the new frame, replacing the contents of the other original frame. The replaced windows remain listed in the Window Directory for convenient redisplay. The Message window is not in a frame, so it cannot be joined.

To join a frame to the frame below:

1. Place the cursor in the frame you want to enlarge. If the frame contains more than one window—for example, a major window and a Command window—you can put the cursor in any window in the frame.
2. Press `Window` - `J`.

The frame containing the cursor is joined with the frame below it, if any; otherwise, it is joined to the frame above it.

To join a frame to the frame above:

1. Place the cursor in the frame you want to enlarge.
2. Press `Window` - `Delete`.

The frame containing the cursor is joined with the frame above it, if any; otherwise, it is joined to the frame below it.

In Figure 4-5, `Window` - `Delete` is used to join frame B with frame C to form a single frame. Because the cursor is in frame B, the windows in this frame expand to occupy the resulting frame.

Joining frames reduces the number of frames on the screen—for example, from three to two in Figure 4-5. The third frame will return the next time you redisplay a window or use a traversal operation to display an object.

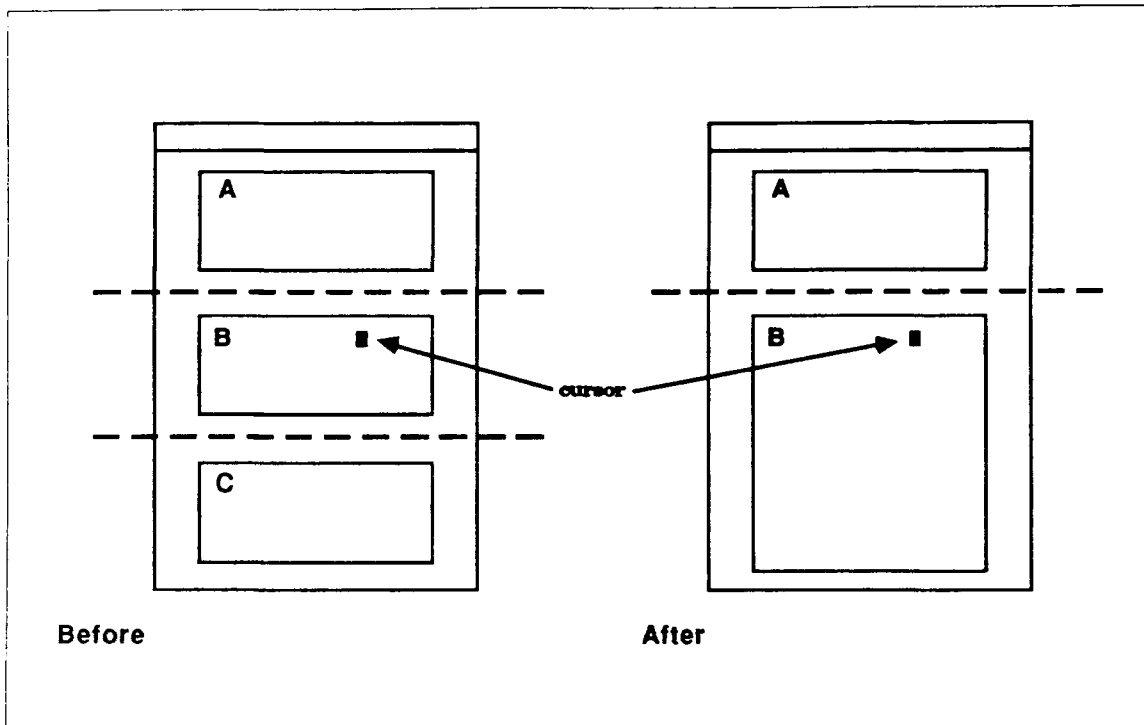


Figure 4-5. Before and After Joining Two Frames

Expanding Windows

You can expand any window to display a greater portion of the object in that window. It can be especially useful to expand Command windows, for reasons given in “Ada Usage in Command Windows,” in Chapter 5. Unlike joining two frames, expanding a window does not necessarily reduce the number of windows on the screen.

To expand a window:

1. Put the cursor in the window you want to expand.
2. Press `Window` - `]`.

The window containing the cursor expands by four lines.

By default, a window expands four lines at a time. You can use the numeric keypad to specify a different number of lines. For example, pressing `numeric 7` - `Window` - `]` expands the window by seven lines.

You can expand any window, including the Message window. When you expand a major window, the frame containing it expands, causing an adjacent frame to shrink. However, when you expand a Command window or a minor window, the other windows within the same frame shrink as much as possible before the overall frame size is adjusted to accommodate the expansion.

Shrinking Windows

When you do not need to view as much of an object but you still want it on your screen, you can shrink its window down to a minimum size of two lines. When you shrink a major window, the frame that contains it shrinks and an adjacent frame expands accordingly. When you shrink a Command window or a minor window, the frame size stays the same and the other windows in the same frame expand to compensate for the shrinkage.

To shrink a window:

1. Put the cursor in the window you want to shrink.
2. Press `Window` - `.`. Use the period key on the main keyboard.

The window containing the cursor shrinks by four lines.

By default, a window shrinks four lines at a time. You can use the numeric keypad to specify the different number of lines. For example, pressing `numeric 7` - `Window` - `.` shrinks the window by seven lines.

Making Frame Sizes Equal

After expanding, joining, and shrinking windows, you can readjust the variously sized frames in a single operation so that they once again divide the screen equally. As a result, windows expand or shrink to fit the equally sized frames. To reset window sizes:

Press `Window` - `Format`. The cursor can be in any window.

All frames are realigned so that they are of equal size.

The `Window` - `Format` operation is based on the current number of frames. Therefore, pressing `Window` - `Format` after joining windows adjusts the remaining frames to be of equal size. Note that the `Window` - `Format` operation does not restore the original number of frames; it simply resizes existing frames. (The full number of frames is restored the next time you display an object or redisplay a window.)

Removing Windows from the Screen

When you no longer want to view the object in a particular window, you can simplify your screen by deleting that window.

To delete a window from the screen:

1. Place the cursor in the window you want to delete.
2. Press `Window` - `D`.

The designated window is removed from the screen.

If you delete a major window, any attached Command windows are also deleted. In contrast, if you delete a Command window, the remaining windows within the frame expand to compensate for the deletion. Deleting the last window in a frame leaves empty screen space, without affecting the sizes of the surrounding frames. You can use the `Window - Format` operation to resize the remaining frames.

When you delete a window using `Window - D`, the deleted window remains listed in Window Directory, where it can be redisplayed easily. However, if you are finished with an object in a window and have no interest in redisplaying it during your session, you can:

- *Release* the object using `Object - X`. Releasing an object deletes its window from the screen and from the Window Directory, saving any changes, if necessary.
- *Abandon* an object using `Object - G`. Abandoning an object releases it without saving changes.

Releasing or abandoning objects can help keep your Window Directory smaller, making it easier to locate entries listed there. Releasing or abandoning objects has other effects; see “Write Locks,” in Chapter 7.

Rearranging Windows on the Screen

Besides being resized and deleted, windows can be rearranged. That is, if you want to change the top-to-bottom order of the major windows on the screen, you can transpose, or exchange, the contents of any two adjacent frames until your major windows appear in the desired order. Transposing wraps around the screen so that you can transpose the contents of the top and bottom frames. The Message window is not in a frame, so transposing ignores it.

Note that you cannot transpose windows *within* a frame. That is, you cannot change the relative position of a major window and a Command window that is attached to it. Rather, you can only transpose windows *between* frames, relocating major windows along with any attached windows.

To transpose two frames:

1. Place the cursor in the lower of the two frames you want to transpose. If the lower frame contains more than one window—for example, a major window and a Command window—you can put the cursor in any window in the frame.
2. Press `Window - T`.

The contents of the current frame are transposed with the contents of the frame above it. The cursor is left in the lower of the two frames.

Figure 4-6 represents the use of `Window - T` to exchange the windows in frame B with the windows in frame A. The cursor remains in frame B. If the cursor had been in the top frame (frame A), pressing `Window - T` would exchange the windows in frames A and C, since transposing wraps.

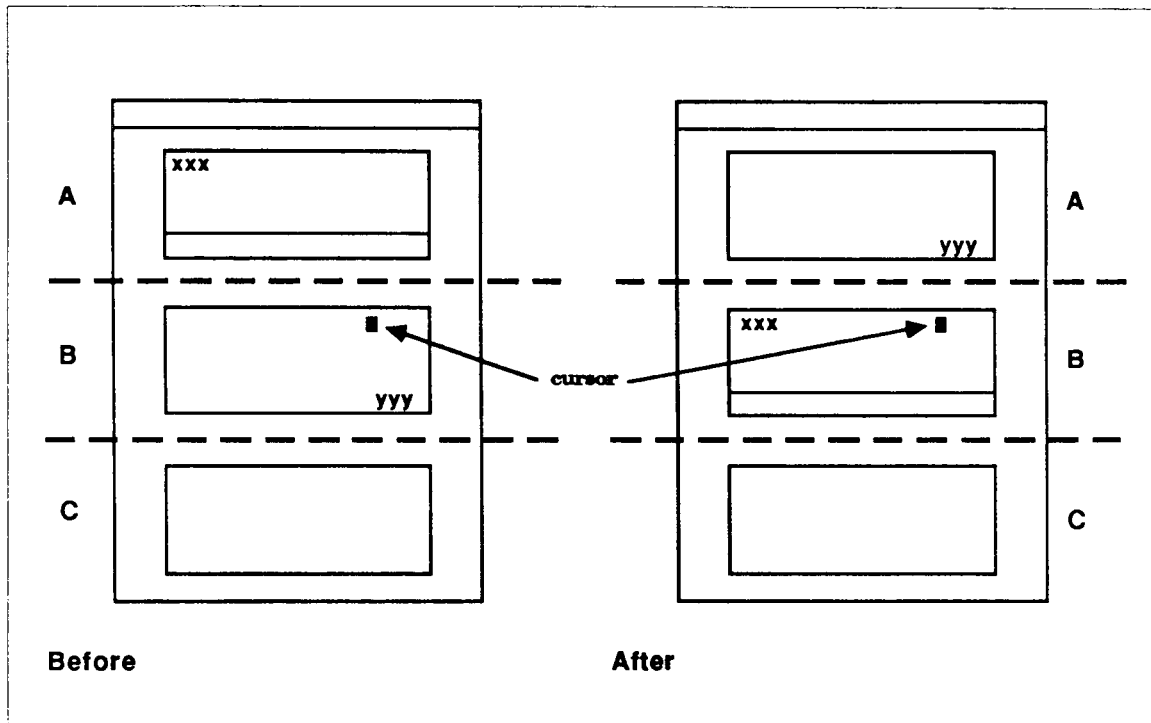


Figure 4-6. Before and After Transposing Windows

Changing the Number of Frames

You can display more or fewer objects on the screen at a time by changing the number of frames into which the screen is divided. The `Window.Frames` command specifies the maximum number of frames that the screen can contain before existing windows are replaced by new windows.

For example, to increase the number of frames to four:

1. Press `Create Command` to open a window in which you can enter a command.
2. In this window, enter the following command:

```
window.frames(4);
```

3. Press `Promote`.

The fourth frame is created the next time a window is opened (for example, the next time you traverse to an object or display a window from the Window Directory).

4. You can equalize the frame sizes by pressing `Window - Format`.

To decrease the number of frames back to three:

1. Press **Create Command** to open a window in which you can enter a command.
2. In this window, enter the following command:

```
window.frames(3);
```

3. Press **Promote**.

The fourth frame remains until you delete a major window or join two frames. From then on, only three frames will be used.

4. You can equalize the frame sizes by pressing **Window** - **Format**.

The number of frames set by the `Window.Frames` command stays in effect until you log out (unless, of course, you change it again). You can change the number of frames more permanently by changing the values for the `Window_Frames` and `Window_Frames_Startup` session switches. Changing the value of these switches changes the default number of frames, which stays in effect across subsequent logins; see the Session and Job Management (SJM) book of the *Rational Environment Reference Manual*.

RATIONAL

Chapter 5. Executing Commands

Your interactions with the Environment are carried out through Environment commands. Using Environment commands, you can invoke operations that create, display, and modify objects, manage windows, display information, and the like. The Reference Summary, in Volume 1 of the *Rational Environment Reference Manual*, provides a complete list of Environment commands.

There are two basic mechanisms for executing Environment commands. You can execute commands by entering them in Command windows; for example, “Logging Out,” in Chapter 2, shows you how to log out by entering the Quit command in a Command window. In addition, many commonly used commands are bound to key combinations; pressing the appropriate key combinations executes these commands. For example, the traversal operations presented in Chapter 3 are invoked by commands bound to key combinations. The Basic Keymap, in the *Rational Environment Basic Operations*, provides a list of those Environment commands that are bound to key combinations. For a more detailed list, see the Keymap in the Reference Summary, Volume 1 of the *Rational Environment Reference Manual*.

An important feature of the Environment is that it uses the Ada programming language as its command language. The Environment commands listed in the Reference Summary are thus predefined Ada procedures and functions that are provided for your use. As with user-created Ada programs, Environment commands are compiled and executed when you enter them through Command windows or key combinations. With Ada as the command language, you do not have to learn a separate grammar for a system-specific command language.

The following sections describe how to:

- Use Command windows
- Get assistance for completing command name fragments and parameter profiles
- Reuse Command windows
- Modify and reexecute commands in a Command window
- Change the default values of commands that are bound to keys
- Take advantage of using Ada as the Environment command language

Using Command Windows

All Environment commands, including commands that are bound to key combinations, can be executed from a Command window. In fact, commands that are not bound to keys can be executed only from a Command window.

To illustrate the basic Command window mechanisms, consider the Definition command. The default form of the Definition command uses the cursor's position to determine which object to display. You have already seen one way of executing the Definition command in its default form—namely, by positioning the cursor and pressing `Definition`. You can perform the same operation using a Command window.

For example, if you are viewing the world `!Users.Anderson.Tools`, you can display the specification for package `String_Tools` as follows:

1. Within the window for `!Users.Anderson.Tools`, position the cursor on the entry for the `String_Tools` package specification, as shown in Figure 5-1:

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
~Rational (Delta) ANDERSON S_1
-----
!Users.Anderson.Tools : Library (World):
  Math_Tools      : C Ada (Pack_Spec);
  Math_Tools      : C Ada (Pack_Body);
  Scan_Tools      : C Ada (Pack_Spec);
  Scan_Tools      : C Ada (Pack_Body);
  String_Tools    : C Ada (Pack_Spec);
  String_Tools    : C Ada (Pack_Body);
= !USERS ANDERSON TOOLS (library) World
```

Figure 5-1. Before Creating a Command Window

2. Press `Create Command` to open a two-line Command window below the window containing `!Users.Anderson.Tools` (see Figure 5-2). The Environment places the cursor in the Command window and remembers the original cursor position in the image of `!Users.Anderson.Tools`.

Observe that the Command window contains a reverse video *prompt* that contains the text `[statement]`, followed by the word `end;`. The prompt indicates where to enter commands (or other Ada statements), and the cursor is placed on the first character of this prompt. The word `end;` is part of the Ada block statement that is supplied in each Command window so that Ada statements (such as Environment commands) can be executed.

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
```

```
= Rational-(Delta)-ANDERSON $_1
-----
!Users Anderson Tools : Library (World):
Math_Tools : C Ada (Pack_Spec);
Math_Tools : C Ada (Pack_Body);
Scan_Tools : C Ada (Pack_Spec);
Scan_Tools : C Ada (Pack_Body);
String_Tools : C Ada (Pack_Spec);
String_Tools : C Ada (Pack_Body);
= !USERS ANDERSON TOOLS (library) world *
[statement]
end;
```

Figure 5-2. After Creating a Command Window

3. Leaving the cursor on the [statement] prompt, enter the Definition command as shown in Figure 5-3. Note that the prompt disappears as you type. You can enter the command in lowercase letters; case is ignored.

You can correct typing mistakes using text editing operations. Some useful operations are summarized in “Correcting Typing Errors,” below.

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
```

```
= Rational-(Delta)-ANDERSON $_1
-----
!Users Anderson Tools : Library (World):
Math_Tools : C Ada (Pack_Spec);
Math_Tools : C Ada (Pack_Body);
Scan_Tools : C Ada (Pack_Spec);
Scan_Tools : C Ada (Pack_Body);
String_Tools : C Ada (Pack_Spec);
String_Tools : C Ada (Pack_Body);
= !USERS ANDERSON TOOLS (library) world
definition
end;
```

Figure 5-3. Entering the Definition Command

4. Press **Promote** to execute the command. At this point, the Environment compiles, links, and then runs the Ada block statement in the Command window.

As shown in Figure 5-4, the command name in the Command window becomes a reverse video prompt after it has been successfully compiled and linked. (The purpose of the prompt is explained in “Reusing Command Windows,” below.) Furthermore, as an Ada statement, the command is displayed in formatted, syntactically complete form—it is capitalized and followed by a semicolon.

If processing the command takes longer than a few seconds, the word ...running appears in the Message window banner. If you attempt to use the keyboard while the ...running message is displayed, your keystrokes are remembered and the requested action is taken only after the message disappears.

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Logo ANDERSON $ _1      running
-----
!!Users Anderson Tools : Library (World):
  Math_Tools : C Ada (Pack_Spec);
  Math_Tools : C Ada (Pack_Body);
  Scan_Tools : C Ada (Pack_Spec);
  Scan_Tools : C Ada (Pack_Body);
  String_Tools : C Ada (Pack_Spec);
  String_Tools : C Ada (Pack_Body);
= !USERS ANDERSON TOOLS (library) : world
  Definition;
end;
```

Figure 5-4. While the Command Is Executing

The ...running message remains in the Message window banner until the Definition command finishes executing. At this point, the String_Tools package specification is displayed in another frame, as shown in Figure 5-5.

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Rational (Logo) ANDERSON $ _1
-----
!!Users Anderson Tools : Library (World):
  Math_Tools : C Ada (Pack_Spec);
  Math_Tools : C Ada (Pack_Body);
  Scan_Tools : C Ada (Pack_Spec);
  Scan_Tools : C Ada (Pack_Body);
  String_Tools : C Ada (Pack_Spec);
  String_Tools : C Ada (Pack_Body);
= !USERS ANDERSON TOOLS (library) : world
  Definition;
end;
```

```
Package String_Tools is
  function Equal (String1 : String; String2 : String;
                 Ignore_Case : Boolean := True) return Boolean;
  function Greater_Than (String1 : String; String2 : String;
                        Ignore_Case : Boolean := True) return Boolean;
  function Less_Than (String1 : String; String2 : String;
                     Ignore_Case : Boolean := True) return Boolean;
  function Lower_Case (S : String) return String;
  function Upper_Case (S : String) return String;
= !USERS ANDERSON TOOLS (STRING_TOOLS V.6) (ada) : Coded
```

Figure 5-5. After the Command Has Executed

Unrecognised Commands

Step 4 in the previous example shows what happens when the Environment is able to compile and link a command successfully. If, however, you enter a command that is not recognized by the Environment, the following occur after you press **Promote**:

- A message is displayed in the Message window.
- The command name remains displayed in normal video, but underlined, in the Command window.

For example, if you enter a misspelled command name, such as “defition,” your screen will look something like this:

```
Semantic error(s), command aborted
=> Rational (Delta) ANDERSON S_1
-----
|Users Anderson Tools : Library (World):
|Math_Tools      : C Ada (Pack_Spec);
|Math_Tools      : C Ada (Pack_Body);
|Scan_Tools      : C Ada (Pack_Spec);
|Scan_Tools      : C Ada (Pack_Body);
|String_Tools    : C Ada (Pack_Spec);
|String_Tools    : C Ada (Pack_Body);
|=>USERS:ANDERSON TOOLS (library)   world
| Defition:
end;
```

Figure 5-6. When a Command Is Not Recognized

If you identify an error in the underlined command name, you can:

1. Make the necessary corrections (see “Correcting Typing Errors,” below).
2. Press **Promote** again to execute the corrected command.

If you cannot find an obvious spelling error and the command still is not accepted, press **Explain** to display further information about the error in the Message window. If the resulting message indicates that the command you entered is undefined, then:

- You may be thinking of a command with a similar name. Check the Reference Summary to verify that the command you entered is in fact defined.
- Some command names contain several name components (see “Environment Commands,” below). Make sure you have entered all the necessary components of the command name.
- It may be that the command you entered is not visible based on the way Ada names are resolved in Command windows. This should not be a problem for the standard set of Environment commands (see “Environment Commands,” below).

Correcting Typing Errors

As you enter a command in a Command window, you can use text editing operations to correct typing errors. You can use these operations to make changes either before you press `Promote` or after a command name has been underlined. Note that if you try to make changes to a command name that is displayed as a prompt, the prompted name will disappear (see “Modifying and Reexecuting Commands,” below). You can redisplay a prompted name that disappeared by pressing `Object` - `U` (see “Recalling Previous Commands,” below).

Table 5-1. Some Useful Editing Operations in Command Windows

Key	Operation
<code>--</code>	Moves the cursor one character to the left.
<code>-</code>	Moves the cursor one character to the right.
<code>Begin Of</code> , <code>End Of</code>	Moves the cursor to the beginning or end of a line.
Any character	Inserts the character to the left of the cursor.
<code>Control</code> <code>D</code>	Deletes the character the cursor is on.
<code>Delete</code>	Deletes the character to the left of the cursor.
<code>Control</code> <code>K</code>	Deletes from the cursor to the end of the line.
<code>Line</code> - <code>Delete</code>	Deletes from the cursor to the beginning of the line.
<code>Line</code> - <code>D</code>	Deletes an entire line.

Canceling Command Execution

After you press a key combination that is bound to a command or after you press `Promote` with a command in a Command window, the Environment compiles, links, and executes the command. You can interrupt and kill this process by taking the following step:

Press `Meta` `G` to stop command processing and execution. Note that the ...running message is turned off so that you regain use of the keyboard.

Command Windows and Attached Windows

A Command window bears a special relationship to the window from which it was created and to which it is attached. For example, many Environment commands, such as the default form of the Definition command, are sensitive to the cursor's position in an image. When such a command is bound to a key, the command uses the cursor's actual position at the time the key is pressed. However, as shown above, when such a command is executed from a Command window, the command uses the cursor's most recent position in the attached window. That is, the command executes as if the cursor is still in the attached window, rather than in the Command window.

Furthermore, as shown in the next section, many commands accept pathnames as parameter values. When such a command is executed from a Command window, the current context for resolving pathnames is determined by the object in the window to which the Command window is attached.

Ada Usage in Command Windows

The use of Ada as the command language provides a number of possibilities for entering commands in Command windows. That is, because Command windows contain Ada block statements and because Environment commands are entered as Ada procedure call statements:

- The rules of Ada syntax allow alternative ways to specify parameter values for those commands that have parameters.
- You can declare variables, constants, and the like within Command windows, and you can build multiple line programs.

Entering Parameters

To illustrate some Command window possibilities, consider the Definition command in greater detail. You can pass information to the Definition command through three parameters, which are shown in the specification for the command:

```
procedure Definition (Name      : String := "<CURSOR>";
                    In_Place  : Boolean := False;
                    Visible   : Boolean := True);
```

These three parameters are characterized briefly as follows:

Name	Requires a string that names the desired object. The default string "<CURSOR>" displays the object indicated by cursor position. See "Special Names and Parameter Placeholders," below.
In_Place	Requires a Boolean value to specify where the requested object is displayed. The default value, false, means that the object is displayed in the least recently visited frame. (Note that <code>Definition In Place</code> is bound to the Definition command with the In_Place parameter set to true.)
Visible	Requires a Boolean value to specify which part of an Ada unit (the visible part or the body) is displayed if the specified name is ambiguous. The default value, true, means that the specification is displayed.

By Ada syntax rules, when you enter a command with parameters, you must:

- Enclose the parameters in a set of parentheses following the command name.
- Separate multiple parameters with commas.
- Enclose each string value, such as the name of an Environment object, in quotation marks.

You can use positional parameter association, which means that parameters must be listed in the order in which they are given in the command specification. You can also use named parameter association, which allows you to list parameters in any order by explicitly naming the formal parameter. Finally, you can omit parameters for which you want to use default values (however, you must use named parameter association to specify any parameters that follow an omitted parameter).

You have already seen the Definition command entered using the default parameters as follows:

```
Definition;
```

Alternatively, you can supply a value for the Name parameter to request an object by name rather than by cursor position. That is, when the current context does not contain the object you want to display, you can use the Name parameter to traverse to that object directly without displaying all intervening objects in the library hierarchy.

Because the name you specify is resolved relative to the current context, you can give all or part of an object's pathname, depending on where the Command window is attached. For example, if the cursor is in the window containing the world !Users.Anderson, you can display the package specification for !Users.Anderson.Tools.String_Tools by creating a Command window and entering the following command:

```
Definition ("tools.string_tools");
```

If, however, the cursor is in a window containing the file !Users.Miyata.Memo, you can create a Command window there and enter the command using a fully qualified pathname:

```
Definition ("!users.anderson.tools.string_tools");
```

In the following command, the first two parameters are specified, so that the named object is displayed in the current frame instead of the least recently visited frame:

```
Definition ("!users.anderson.tools.string_tools", true);
```

However, you must use named parameter notation to omit the first parameter and specify the second, as in the following command. This command finds the object indicated by the cursor and displays it in the current frame:

```
Definition (In_Place => true);
```

Note that the parameter value true is an Ada identifier and therefore is not enclosed in quotation marks.

Using the Command Window Declare Block

Each Command window contains a complete Ada block statement, as shown in the expanded Command window in Figure 5-7. (Details about the *use* clause are explained in "Environment Commands," below.) To reveal the block statement in a Command window, press **Window** - **]** several times, and then press **Image** - **Begin Of**.

```

declare
  use Editor, Library, Common;
begin
  [statement]
end;
```

Figure 5-7. The Ada Block Statement in a Command Window

You can write declarations within the declarative part of the block, between *declare* and *begin*. For example, the following command is another way to display the package specification for `!Users.Anderson.Tools.String_Tools`. In this Command window, a constant is declared and used in an expression to produce a string value for the *Name* parameter:

```

declare
  use Editor, Library, Common;
  Name_Prefix : constant string := "!users.anderson.tools";
begin
  Definition (Name_Prefix & ".string_tools");
end;
```

One or more Environment commands (or other Ada statements) can be put in the body of the block statement, between *begin* and *end*. Multiple statements must be separated by semicolons. For example, the following displays the Window Directory, searches for (and puts the cursor on) an asterisk (which indicates a modified image), and then displays the window indicated by the cursor:

```

declare
  use Editor, Library, Common;
begin
  Window.Directory;
  Editor.Search.Next("*");
  Definition;
end;
```

Getting Prompting Assistance

The Environment provides several kinds of assistance for entering commands—namely, formatting and semantic completion. Formatting is the process of verifying and, if necessary, completing the command syntax according to Ada syntax rules. Once the command syntax is considered complete, the formatted command is pretty-printed. Semantic completion, in contrast, is based on the resolution of Ada names, supplying a completed name and a parameter profile for unambiguous command name fragments.

Getting Formatting Assistance through the Promote Key

When you press `Promote` to execute the contents of a Command window, the block statement in the window is formatted as the first step in the compilation process. Therefore, when entering a command, you can omit redundant syntactic elements such as final punctuation. For example, you can enter a command as follows and then press `Promote`:

```
definition("!users.anderson.tools.string_tools
```

When the command has been compiled and linked, it will be displayed as a prompt in the Command window, formatted like this:

```
Definition ("!users.anderson.tools.string_tools");
```

Getting Semantic Completion through the Complete Key

Before pressing `Promote`, you can get semantic completion by pressing `Complete`. Pressing `Complete` after entering an unambiguous command name fragment provides a completed command name and a parameter profile, if any. `Complete` also formats the contents of a Command window, inserting the required punctuation and performing pretty-printing.

For example, you can get semantic assistance for the Definition command as follows:

1. In a Command window, enter a fragment of the command name, such as the following: `defi`
2. Press `Complete` (located next to `Promote`).

After a pause, the full command name appears in the Command window, along with the parameter profile in named parameter notation. Note that the command is formatted, the default parameter values appear as prompts, and the cursor is placed on the first prompt:

```
Definition (Name => <CURSOR>, In_Place => false, Visible => false);
```

3. At this point, you can fill in the parameter prompts as desired (see "Filling In Parameter Prompts," below) and press `Promote` to execute the command.

Note that you can use `Complete` to provide semantic assistance after you enter a partial or a complete command name. However, `Complete` cannot provide assistance if you have supplied any parameters or parameter syntax such as the left parenthesis: `(`. In other words, `Complete` cannot provide a partial parameter profile, only a whole one.

Filling In Parameter Prompts

As shown in the Definition example above, when the parameter profile for a command is displayed, each parameter name is associated with a reverse video prompt. The prompt for a given parameter displays the default parameter value, if there is one. Otherwise, the prompt includes the parameter's type in brackets. For example, the first parameter of the Io.Create command is of type File_Type, as indicated by the prompt in the following:

```
Io.Create (File => FILE_TYPE_PROMPT, ...
```

Parameter prompts provide you with several alternatives:

- You can use the supplied default parameter value, if any. To do this:
 1. Leave it in prompt form. Do not type on the prompt.
- You can replace the supplied value with another value. To do this:
 1. Put the cursor on the appropriate prompt. You can use the cursor keys to move the cursor between prompts or you can move the cursor directly (see "Moving between Prompts," below).
 2. Enter the desired value. As usual, the prompt disappears when you start to type on it.

Note that prompts for string values already include quotation marks, which remain after the prompt disappears, so that you do not have to type the enclosing quotation marks.
- You can modify the supplied value to create a new value. To do this:
 1. Put the cursor on the appropriate prompt.
 2. Press `[Item Off]` to turn off the prompt and display the supplied value as regular text. Now the value will not disappear when you type on it.
 3. Use text editing operations to modify the value as desired.

Moving between Prompts

You can move directly from one prompt to the next without repeatedly pressing the cursor keys as follows:

- Press either `[Next Item]` or `[Next Prompt]` to move the cursor to the next prompt.
- Press either `[Previous Item]` or `[Previous Prompt]` to move the cursor back to the previous prompt.

Note that `[Next Item]` and `[Previous Item]` can be used to move between underlined items as well as prompted ones.

Completing Ambiguous Name Fragments

In the sample completion of the Definition command above, the correct command name is found because the name fragment “defi” is resolved uniquely—that is, only one command name contains the string “defi”. If you supply a name fragment that is shared by multiple commands, displays a list of those commands. You can decide which command in the list is correct, modify the name fragment in the Command window accordingly, and press again.

For example, pressing to complete the name fragment “de” produces a display like the one in Figure 5-8. As shown, completion opens a window, called a *menu window*, which displays a list of Ada specifications that contain the given name fragment. Note that the cursor remains in the Command window.

```
Rational Environment
  D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
= Rational (Delta) ANDERSON S_1
-----
!Users Anderson : Library (World):
Calculation      : C Ada (Pack_Spec);
Calculation      : C Ada (Pack_Body);
Documentation     : Library (Directory);
Factorial        : C Ada (Func_Spec);
Factorial        : C Ada (Func_Body);
Login            : C Ada (Proc_Spec);
-----
= ANDERSON (library) world
begin
  De
-----
List of possible completions
De =>
  procedure Def (Name : String := "<CURSOR>";
                In_Place : Boolean := False; Visible : Boolean := True);
  procedure Definition (Name : String := "<CURSOR>";
                       In_Place : Boolean := False; Visible : Boolean := True);
  procedure Demote;
-----
= (menu)
```

Figure 5-8. A List of Possible Completions for the Name Fragment “De”

From the menu, you can determine that “defi” is the minimal string that uniquely identifies the Definition command. At this point, you can change the name fragment in the Command window to “defi” and press again.

Completion Menu Entries

Completion tries to resolve name fragments by searching through visible Ada declarations. Therefore, entries in the completion menu correspond to Ada declarations, which include:

- Compilation units that are defined directly in the library hierarchy
- Declarations that are nested within compilation units

Furthermore, a completion menu can contain entries for user-defined Ada units in addition to predefined Environment commands. Finally, since name resolution depends on what is visible in the current context, the menu for a given name fragment can contain different entries, depending on where the Command window was created.

When the menu for a fragment is first displayed, each entry is the specification of a possible completion of the name fragment, as shown in Figure 5-8. In the case of the Definition command, the specification in the menu provides enough information for you to make the command name fragment unambiguous. Sometimes this information will not be sufficient. When this is the case, you can expand the menu entries to get the full pathnames of each of the specifications in the menu.

For example, completing the name fragment “de” produces a completion menu containing the following entries:

List of possible completions

```
De =>
  procedure Def (Name : String := "<CURSOR>";
    In_Place : Boolean := False; Visible : Boolean := True);
  procedure Definition (Name : String := "<CURSOR>";
    In_Place : Boolean := False; Visible : Boolean := True);
  procedure Demote;
```

To expand these menu entries:

1. Move the cursor into the menu window. The cursor can be anywhere in the window.
2. Press `Object` - `F` to display the fully qualified Environment pathnames for all entries, as shown:

List of possible completions

```
De =>
  procedure !COMMANDS.ABBREVIATIONS.DEF (Name : String := "<CURSOR>";
    In_Place : Boolean := False; Visible : Boolean := True);
  procedure !COMMANDS.COMMON.DEFINITION (Name : String := "<CURSOR>";
    In_Place : Boolean := False; Visible : Boolean := True);
  procedure !COMMANDS.COMMON.DEMOTE;
```

If you still want more information about a particular entry, you can position the cursor on the entry and press `[Definition]`. The Ada specification for the command is displayed in a separate window.

When you have decided which command you want:

1. Return the cursor to the Command window.
2. Enter the correct, fully qualified Ada name for the command. Note that the expanded menu entries do not show the form of the command as you would actually enter it. That is, you cannot invoke a command from a Command window by entering its fully qualified Environment pathname or its attributes (for example, `!Commands.Common.Definition`). See “Environment Commands,” below, for a discussion of command names.
3. If necessary (when names are overloaded), enter the command’s parameters (without completion assistance) to make the command unambiguous.
4. Execute the command.

For more information about menus, see “Menus” in the Editing Specific Types (EST) book of the *Rational Environment Reference Manual*.

Abbreviating Commands

For your convenience, the Environment recognizes abbreviated forms for a number of commands. Each of the abbreviated command names is defined by an Ada procedure in the world `!Commands.Abbreviations`.

For example, the menu in Figure 5-8 contains an entry for `Def`. The expanded form of this entry—namely, `!Commands.Abbreviations.Def`—indicates that `Def` is indeed an abbreviation. To verify that `Def` abbreviates the `Definition` command, you can display the body for the `Def` procedure and note that it calls the `Definition` command.

By creating procedures similar to those provided, your project team can define projectwide abbreviations in `!Commands.Abbreviations`.

Clearing a Command Window

If you find you have entered a command incorrectly, or if semantic completion supplies a command other than the one you wanted, you may find it easier to start over from a fresh Command window rather than making corrections using editing operations.

To clear a Command window:

1. Put the cursor in the Command window you want to clear.
2. Press `[Edit]`.

A fresh block statement and `[statement]` prompt are displayed.

After clearing a Command window, you can recall its previous contents; see “Recalling Previous Commands,” below.

Reusing Command Windows

After commands have been executed, Command windows remain available for further use. As long as a Command window is displayed, you can execute another command in it, modify and reexecute the same command, or recall and reexecute previous commands.

Note that `[Create Command]` helps you reuse existing Command windows. If the window containing the cursor already has a Command window, then pressing `[Create Command]` moves the cursor into it, instead of creating an extra one.

Executing Subsequent Commands

Recall from “Using Command Windows,” above, that commands turn into prompts after they have been successfully compiled and linked. By turning into prompts, previously entered commands disappear when you type on them. This allows you to enter subsequent commands without having to explicitly delete the previous command or create a new Command window.

To use the same Command window for entering another command:

1. Move the cursor into the Command window.
2. With the cursor anywhere on the prompt, enter the new command. The prompt containing the old command disappears.
3. Press `[Promote]`.

Reexecuting the Same Command

Occasionally you may need to execute the same command several times—for example, you can execute the `Queue.Display` command periodically to check on the progress of the printer queue. To reexecute the same command without modifying it:

1. Move the cursor into the Command window containing the command in prompt form.
2. Without typing on the prompt, press `[Promote]`.

Modifying and Reexecuting Commands

You can modify the name or parameters of a previously executed command without having to reenter the entire command. If the command you want to modify is displayed as a prompt, you must use `[Item Off]` to display it as regular text so that it won't disappear when you try to modify it.

For example, say you want to display the specification for the package `!Users-Anderson.Tools.Scan_Tools`, and the following command appears as a prompt in the Command window:

```
begin
  Definition ( !users anderson tools string_tools spec );
```

You can change `string` to `scan` and reexecute the command as follows:

1. Move the cursor into the appropriate Command window.
2. With cursor on the prompt, press `[Item Off]`. This turns off the reverse video prompt, leaving the text of the command in the regular screen font.
3. You can now use text editing operations to change the word `string` to `scan`.
4. Press `[Promote]` to execute the modified command.

Another approach to modifying commands is to use `[Complete]` to display just the parameter values as prompts. This makes it easy to change one or more parameter values without losing the rest of the command. You can use `[Complete]` with commands that appear as prompts or with commands that appear as regular text.

For example, assume that the `Definition` command appears in a Command window as a prompt, as shown above, and you want to display the body of the procedure `!Machine.Editor_Data.Rational_Commands`. If you were to turn the entire command into regular text, as shown above, you would have to delete most of the old parameter value before entering the new value, since the two values have so little in common. A more convenient alternative is to leave just the parameter value as a prompt so that it will disappear when you type on it. To do this:

1. With the cursor anywhere in the Command window, press `[Complete]`.

The existing parameter value is turned into a prompt, leaving the rest of the command in regular text, as shown:

```
begin
  Definition ( !users anderson tools string_tools spec );
```

2. You can now type the new parameter value. Note that the quotation marks from the previous value remain.
3. Press `[Promote]` to execute the modified command.

Note that when `[Complete]` is used to modify previously executed commands, it changes only the given parameters into prompts; it does not supply prompts for any parameters that were omitted when the command was executed.

Recalling Previous Commands

Each Command window maintains a history of what was executed in it since it was created. This history allows you to redisplay, edit, and reexecute previously executed commands.

A Command window history is maintained as a series of Ada block statements. Each time you press `Promote` or `Complete`, the current contents of the Command window are added to the series. Furthermore, using `Edit` to clear a Command window simply adds a block statement containing a `[statement]` prompt into the series. It is important to emphasize that it is not just the command statements that are remembered, but the entire block, including anything you declared in the block's declarative portion.

You can step through the history series to redisplay any of the remembered block statements as follows:

- Press `Object` - `U` to step backward in the series, toward the original block that contains the `[statement]` prompt. `U` stands for "undo."
- Press `Object` - `R` to step forward in the series, toward the more recently executed commands. `R` stands for "redo."

Once you have redisplayed a block statement from the Command window history, you can modify and then reexecute it. In addition to using text editing operations, you can press `Complete` to change parameter values into prompts. When you modify and reexecute an intermediate block statement in a history series, the modified block is inserted into the series at the intermediate point. Thus, the most recently executed command is not necessarily in the last block in the history series.

A Command window history is preserved even if you use `Window` - `D` to delete the Command window or the window to which it is attached. In fact, if you delete a Command window and then press `Create Command` again, the deleted Command window is reopened with the same contents and the same history. However, a Command window history is destroyed if you use `Object` - `G` or `Object` - `X` to delete the Command window or its associated major window.

Keys and Command Windows

So far, you have used key combinations that execute commands directly, using default parameter values. Certain key combinations (for example, `Create Text`) do not use default values automatically; instead, they prompt you for values by opening a Command window containing the command you want to execute. You can then fill in the prompts and press `Promote`.

You can use `Prompt For` to cause any key combination to open a Command window containing the associated command. For example, if you want to use nondefault values in the Definition command, you can use the following shortcut to display the command in a Command window:

1. Press `Prompt For`.
2. Press `Definition`.

A Command window is opened containing the Definition command with prompts for all its parameters.

Environment Commands

As Ada units, the predefined Environment commands have specifications and bodies. The command specifications are available for you to inspect; however, the bodies are provided only in object code form.

Environment command specifications are declared in Ada packages that reside in Environment worlds. A number of key packages are defined in the world `!Commands`, including `Ada`, `Common`, `Compilation`, `Debug`, `Editor` and its subpackages, `Job`, `Library`, `Links`, `Switches`, `Text`, and `What`. The world `!Io` also contains packages of frequent use, including packages `Io` and `Text_Io`.

Fully Qualified Ada and Environment Names

It is important to distinguish a command's fully qualified Environment name from its fully qualified Ada name. The latter, which you can enter in a Command window, contains a simple name preceded by a name component for each enclosing unit—for example, `Common.Definition` or `Editor.Cursor.Up`. The former, which locates a command in the Environment, starts with `!` and includes name components for Environment libraries as well as Ada units—for example, `!Commands.Common.Definition` or `!Commands.Editor.Cursor.Up`.

You can always use a command's fully qualified Ada name in a Command window. In some cases, however, you can use a simpler name by virtue of the *use* clause in the Command window. For example, if a Command window *use* clause refers to packages `Editor`, `Library`, and `Common`, you can omit the corresponding name components for commands that are defined in these packages, as shown:

```
declare
  use Editor, Library, Common;
begin
  Cursor.Up;
  Definition;
end;
```

Note that different packages are included in a Command window *use* clause because of context. That is, Command windows opened in a library context contain a *use* clause for packages `Editor`, `Library`, and `Common`, whereas Command windows opened in an Ada unit context contain a *use* clause for packages `Editor`, `Ada`, and `Common`. You can always edit the *use* clause in a Command window to suit your needs.

To help simplify fully qualified Ada names, abbreviated names are available for many of the packages in !Commands, including:

Acl	Access_List
Comp	Compilation
Lib	Library
Q	Queue

Visibility in Command Windows

When the contents of a Command window are compiled, Ada names that are not declared in the Command window itself are resolved using a mechanism called a *searchlist* in place of a formal *with* clause. A searchlist is an ordered list of libraries. When a Command window contains an Ada name to be resolved, the Environment searches through each of these libraries until a unit is found that has the name in question; the found unit is then used for semantic analysis of the Command window contents. More specifically, units that are found through the searchlist are used in constructing for the Command window an implicit *with* clause that provides visibility to names being resolved.

By default, your searchlist includes the current context; therefore, in a Command window attached to a library, you can reference subprograms, packages, and declarations within packages that are defined there. Your default searchlist also contains the libraries you need to execute standard Environment commands. When you create your own programs, you may have to update your searchlist to execute these programs from contexts other than where they are defined. See the Session and Job Management (SJM) book of the *Rational Environment Reference Manual* for further discussion of searchlists. Furthermore, the *Rational Environment Basic Operations* summarizes operations for updating a searchlist.

Special Names and Parameter Placeholders

A final point is that many Environment commands have parameters of type String that default to predefined Environment values called *special names* and *parameter placeholders*. You have already seen an example of a special name in the Definition command:

```
Definition (Name => "<CURSOR>", ...
```

As shown, a special name is enclosed in angle brackets and provides a way of designating an object without having to name it in the command. Two commonly used special names include:

- "<CURSOR>" Refers to the object designated by the cursor's position.
- "<IMAGE>" Refers to the object whose image is the current context.

Other special names are described in the Key Concepts of the Library Management (LM) book of the *Rational Environment Reference Manual*.

Some commands have parameters of type `String` whose default values are given as placeholders. Unlike special names, placeholders do not actually refer to any object. Instead, a given placeholder provides a clue about what you should enter in its place. Therefore, placeholders cannot be used as parameter values; they must be replaced. Following is an example of a placeholder indicated by a double pair of inverted angle brackets:

```
Library.Create_World (Name => ">>WORLD NAME<<", ...
```

Other placeholders are described in the **Key Concepts of the Library Management (LM)** book of the *Rational Environment Reference Manual*.

Chapter 6. Getting Help

The Environment help facility provides a number of ways to get on-line help for Environment resources such as commands and tools. Various help operations are bound to the function keys in the "Help" column of the keyboard overlay. You can use these keys to answer the following questions:

- How does the Environment help facility work?
- What command does this key execute?
- What does this command do?
- What Environment commands and tools pertain to this topic?

In all cases, the requested information is displayed in an Environment window called the Help window.

How Do the Help Keys Work?

To display a description of all of the help keys:

Press `[Help On Help]`. The cursor can be anywhere on the screen.

The Help window is opened, displaying a summary of what each help key does. The Help window is identified in its window banner.

To read the entire summary, you may need to scroll or enlarge the window.

What Command Does This Key Execute?

You can use `Help On Key` to find out what command is bound to a particular key or key combination. For example, to display help for `Definition`:

1. Press `Help On Key`. The cursor can be anywhere on the screen.

The Message window prompts you to press the key or key combination for which you want help:

Press key to be described:

2. Press the desired key or key combination—in this case, `Definition`. (You can press any single key, any modified key combination, or any item-operation key combination.)

The Message window echoes the name of the key you pressed. Because `Definition` is the function key `F10`, the display reads:

Press key to be described: F10

The Help window displays the help message for the command that is bound to the specified key. In this example, the command is `Common.Definition`, as shown in Figure 6-1, in the next section.

Reading Help Messages

Figure 6-1 shows the help message for `Common.Definition` as it appears in the Help window:

```

Press key to be described: F10
-----
COMMANDS.COMMON.DEFINITION is bound to: F10, S_F10, S_RIGHT, CM_RIGHT, OBJECT.

procedure Definition (Name      : String := "<CURSOR>";
                     In_Place  : Boolean := False;
                     Visible   : Boolean := True);

Finds the defining occurrence of the named or designated item and
displays that defining occurrence in a new window.

This procedure finds the location where the item is defined. The
procedure attempts to find the most reasonable definition of the
object, given the current editing context.

Specifically, this procedure has the following effects:

o Ada images: Finds the defining occurrence of the designated
  element and brings up its image in a window on the screen.
-----

```

Figure 6-1. The Help Message for the `Common.Definition` Command

Note that a help message for a command consists of:

- A dashed line indicating the beginning of the message.
- A list of all the key combinations to which the command is bound.
- The specification for the command, including its parameter profile and default parameter values.
- A description of the command extracted from the *Rational Environment Reference Manual*. If the description is long, you can scroll or enlarge the Help window.

The help messages for other topics may have different formats. For example, the help message for a package typically contains introductory material pertaining to all commands in that package.

Each subsequent help message you request is appended to the last message in the Help window. (Thus, the dashed line at the top of a message separates it from the previous message.) The Help window contains all the help messages you request from the time you log in until you log out. To see previous messages, scroll back through the Help window.

Note that if the Help window has been replaced by other Environment windows, you can redisplay it by pressing `[Help Window]`.

What Does This Command Do?

If you know a command's name, you can use the help facility to find out what the command does. For example, to display help for the command `Common.Definition`:

1. Press `[Prompt For] - [Help]`. The cursor can be anywhere on the screen.

A Command window is automatically opened containing the following command:

```
What.Does (Name => "");
```

2. At the parameter prompt, enter the name of the command for which you want help. (Note that help messages are available for packages as well as commands.)

Enter as much of the name as you know—if you can, enter a qualified name such as `common.definition`; otherwise, enter a simple name (`definition`):

```
What.Does (Name => "definition");
```

3. Press `[Promote]`.

If there is no help for the name you entered, a message is displayed in the Message window.

If the name you entered resolves to a single command, the Help window displays the help message for that command. Because both `Definition` and `Common.Definition` resolve uniquely, the help message shown in Figure 6-1 is displayed.

If the name you entered requires further qualification to resolve to a single command, the Help window displays a menu of fully qualified command names for you to choose from. (See “Reading Help Menus,” below.)

What Commands Pertain to This Topic?

You can use the steps given in the previous section to find out what Environment resources are available for a given topic. For example, to find out what commands and tools are available for moving the cursor, you can supply “cursor” as a clue instead of a command name:

1. Press `Prompt For` - `Help` to open a Command window containing the `What.Does` command.
2. At the prompt, enter the topic clue—in this case, `cursor`:

```
What.Does (Name => "cursor");
```
3. Press `Promote`. The index entries in the help facility are searched. As before:
 - If there is no help for the topic you entered, a message is displayed in the Message window.
 - If the clue matches only a single entry, the help message for that entry is displayed.
 - Otherwise, the Help window displays a menu of topics and command names that contain the clue you entered. (See “Getting Further Help from a Menu,” below.)

In this example, entering the word “cursor” displays the menu shown in Figure 6-2, in the next section.

Reading Help Menus

A help menu presents you with choices for getting further help. The Help window displays a help menu when:

- You specify a command name that requires further qualification.
- You specify a topic that matches some portion of more than one command name.

For example, the help menu for the topic “cursor” is shown in Figure 6-2:

```
Prompt For: F11
= Rational |Delta| ANDEFSON S_1
-----
!Users Anderson : Library (World):
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Complex_Numbers  : Library (World);
Display_Factorial : C Ada (Proc_Spec);
Display_Factorial : C Ada (Proc_Body);
Documentation     : Library (Directory);
Factorial        : C Ada (Func_Spec);
Factorial        : C Ada (Func_Body);
= 'USERS ANDEFSON |library| world
  what Does (Name => "cursor")
end;
-----
Index entries related to "cursor"
!Commands Editor.Cursor.Up
!Commands Editor.Cursor.Right
!Commands Editor.Cursor.Previous
!Commands Editor.Cursor.Next
!Commands Editor.Cursor.Left
!Commands Editor.Cursor.Forward
!Commands Editor.Cursor.Down
!Commands Editor.Cursor.Backward
!Commands Editor.Cursor
!Io.Window.Io.Move_Cursor
!Io.Window.Io.Position_Cursor
!Io.Window.Io.Report_Cursor
-----
= Help Window (help)
-----
```

Figure 6-2. The Help Menu for the Topic “Cursor”

Note that the word “cursor” is found in the simple name as well as in other name components of these command names. In general, the help facility matches the topic (or name) you enter against fully qualified command names. The match succeeds if the word you enter is found as a segment of a command name—that is, if a command name contains the specified word between periods or underscores. Thus, “cursor” is found in Move_Cursor and in Cursor.Up. In some cases, partial segments are matched.

Getting Further Help from a Menu

From the help menu, you can:

- Display the help message for a given command or package.
- Traverse to the Environment Ada specification for the command or package.

To display a help message from a help menu:

1. Put the cursor on the entry for which you want help.
2. Press `Explain`.

The appropriate help message is displayed in the Help window following the menu.

To display a help message for a different entry on the menu, you can scroll back to the menu in the Help window and repeat the process for that entry.

To traverse to the Ada specification for an entry on the help menu:

1. Put the cursor on the desired entry.
2. Press `Definition`.

A window is opened containing the appropriate specification.

Part II. Editing Text

Contents

Chapter 7. Creating and Saving Text Files	II-1
Creating Text Files	II-1
Entering Text	II-3
Saving Changes as You Edit	II-4
Versions of Files	II-4
Discarding Changes Since the Last Save	II-4
Opening Existing Files for Editing	II-5
Example 1: Opening a Displayed File	II-5
Example 2: Opening a Selected File	II-5
Write Locks	II-6
Closing Files for Editing	II-7
Chapter 8. Modifying Text	II-9
Adding Text	II-9
Operating on Designated Text Items	II-10
Text Items	II-10
Patterns in Editing Operations	II-11
Selecting Text Items	II-11
Selecting an Arbitrary Stretch of Text	II-11
Using Object Selection	II-12
Selecting within a File's Structural Hierarchy	II-12
Turning Selections Off	II-13
Summary of Selection Operations	II-13
Deleting Text	II-14
Retrieving Deleted Text	II-14
Copying and Moving Text	II-15

Copying Text	II-15
Duplicating a Line	II-15
Moving Text	II-16
Summary of Copy and Move Operations	II-16
Transposing Text	II-16
Searching and Replacing	II-17
Example: Searching for a String	II-18
Summary of Search and Replace Operations	II-19
Controlling Case and Text Format	II-20
Changing Character Case	II-20
Adjusting Text Format	II-21
Setting Word Wrap for Text	II-21
Filling Existing Lines of Text	II-22
Justifying Text	II-22
Centering Lines	II-23
Changing the Fill Column	II-23
Inserting Page Breaks	II-23

Chapter 7. Creating and Saving Text Files

Environment text files are used for composing and storing documentation, memos, nonbinary test data, and the like. This chapter describes the basic Environment operations for:

- Creating a text file
- Entering text
- Saving the text you entered
- Opening existing text files for editing
- Closing text files when you are finished editing them

Operations for modifying text are described in Chapter 8, “Modifying Text.”

Although it is possible to store Ada source code in a text file, doing so means that you will not be able to take advantage of Environment facilities for editing Ada units. To create Ada units, see Part III, “Developing Simple Ada Programs.”

Creating Text Files

Text files exist in libraries, so before you create a text file, you must decide where it will reside. By specifying a fully qualified pathname, you can create a text file in any library from any context. Typically, however, you will create text files directly in a library that you are currently viewing.

For example, assume that you are working in the library `!Users.Anderson.Documentation`, and you want to create a file there called `Research_Notes`. To do so:

1. Put the cursor anywhere in the window containing the appropriate library—in this case, `!Users.Anderson.Documentation`.
2. Press `[Create Text]`. A Command window is opened containing the `Text.Create` command and its parameter profile.
3. With the cursor on the `Image_Name` parameter prompt, enter the name of the file you are creating—in this case, `Research_Notes`. Filenames are strings that have the same syntax as Ada identifiers.

4. Leave the default value for the second parameter, Kind.
5. Press `Promote`. After the command executes, the screen appears as shown in Figure 7-1. Specifically:
 - The directory `!Users.Anderson.Documentation` contains an entry for the text file `Research_Notes`.
 - A window is created containing the new, empty file, and the cursor is placed in this window.
6. At this point, you can enter text (see "Entering Text," below).

```

Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
-----
RATIONAL (Delta) ANDERSON S 1
-----
!Users.Anderson.Documentation : Library (Directory):
History_Log      : File (Text);
Research_Notes  : File (Text);
Status_Report   : File (Text);

-----
RATIONAL (Delta) ANDERSON S 1
-----
!USERS.ANDERSON.DOCUMENTATION (Library) : Directory
Text Create Image_Name => "research_notes", Kind => Text File)
end;

-----
RATIONAL (Delta) ANDERSON S 1
-----
DOCUMENTATION RESEARCH_NOTES V(1) (text)

```

Figure 7-1. After Creating the File `!Users.Anderson.Documentation.Research_Notes`

Note in Figure 7-1 that the window banner for the new file displays the rightmost portion of the file's fully qualified pathname (the name is too long to be displayed entirely). The pathname is followed by an attribute indicating which *version* of the file you are viewing. For a newly created file, the version attribute is 'V(1)', because this is the first version of the file. Versions are described in "Saving Changes as You Edit," below. Following the filename in the banner, the word `(text)` indicates that text editing operations are available for editing this file.

Entering Text

Executing the Text.Create command both creates a file and opens that file for editing. In fact, the window banner of a newly created file contains the blank symbol to the left of the filename to indicate that you can modify the file (see “Modification Symbols in the Window Banner,” in Chapter 4).

Because the file is open for editing, you can enter text simply by typing. As you enter text, bear in mind the following:

- You can type indefinitely to the right or down.
 - You can start a new line by pressing `Return`.
 - You can keep text lines within a specific column width by requesting automatic word wrap; see “Controlling Case and Text Format,” in Chapter 8.
- To enter numeric data, use the number keys on the main keyboard, not on the numeric keypad.
- You can make changes to the text you enter using the operations shown in Chapter 8, “Modifying Text.”

As shown in Figure 7-2, the modification symbol on the banner changes from a blank to an asterisk when you enter text, indicating that the file’s image has been changed but the file itself has not yet been updated. If you try to log out at this point, the Quit command with default parameters will fail because there are unsaved changes.

```
Rational Environment
D_9_25_1 Copyright 1984, 1985, 1986, 1987, by Rational.
* Rational: /Oct/1987 ANDERSON_8_1
-----
!Users_Anderson.Documentation : Library (Directory):
History_Log : File (Text);
Research_Notes : File (Text);
Status_Report : File (Text);
-----
* ANDERSON_DOCUMENTATION.library | Directory
| text:create: /midgc:Name=>research_notes, Kind=>text:file:
end;
-----
Following is a compilation of research done on local Chinese restaurants
that feature inexpensive lunch specials:
-- House of Yee. Lunch special: $3.75

Ordered the Kung Pao Chicken (**1/2), Beef Zucchini (**), and Almond
Chicken (**). Soup of the day was Hot and Sour Soup.
-----
* DOCUMENTATION.RESEARCH_NOTES v1.1: text:
-----
```

Figure 7-2. After Entering Text

Saving Changes as You Edit

You can periodically update the file you are editing by saving the text you have entered. Use `Enter` to save changes and leave the file open for further editing:

1. Put the cursor anywhere in the file's image.
2. Press `Enter`.

The following message appears in the Message window to indicate that the changes were saved, or *committed*:

```
Commit of !USERS.ANDERSON.DOCUMENTATION.RESEARCH_NOTES'V(2) is complete.  
The modification symbol in the banner changes from an asterisk to a blank,  
indicating that changes have been saved and that the file is still open for up-  
dating. You can continue entering or modifying text. At this point, you can log  
out without losing changes.
```

Two other operations can be used to save changes—namely, pressing `Promote` or pressing `Object - X`. These operations have additional effects, which are described in “Closing Files for Editing,” below.

Versions of Files

The message in step 2 above names the saved file with the version attribute 'V(2) because each time you save changes to a file, a new version of the file is created. By default, the Environment retains one version in addition to the current version; other versions are permanently deleted. In the above example, version 2 is now the current version, and version 1 is retained. If you make changes and press `Enter` another time, version 3 becomes the current version, version 2 is retained, and version 1 is permanently deleted.

Previous versions of Environment objects are retained so that you can undo saved changes. You can cause a retained version to become the current version and you can also increase the number of versions that are retained; see the Library Management (LM) book of the *Rational Environment Reference Manual*.

Discarding Changes Since the Last Save

If you decide that you do not want to save your most recent changes, you can discard all unsaved changes by reverting the file to the way it was the last time you saved it. To discard unsaved changes and leave the file open for further editing:

With the cursor anywhere in the file, press `Object - L`.

A Command window is created containing the `Common.Revert` command. Press `Promote` to execute the `Common.Revert` command.

Opening Existing Files for Editing

When you create a file, it is automatically opened for editing, and you can continue to edit it as long as it remains open. However, if you want to modify an existing file that is not already open, you must explicitly open it. (Of course, you can open a file only if you have been granted write access to it.)

Example 1: Opening a Displayed File

Assume that you have used `[Definition]` to view the contents of an existing file called `!Users.Anderson.Documentation.Status_Report`. Upon reading the file, you notice an error. However, you cannot modify `Status_Report` at this point, because `[Definition]` does not open objects for editing. In fact, the window banner for the file contains an equals sign (=), indicating that the file is read-only. If you try to type on the file's image, the following message is displayed in the Message window:

This image is read-only

To open the `Status_Report` file for editing:

1. Put the cursor anywhere in the window containing `Status_Report`.
2. Press `[Edit]`. Note that the banner symbol changes to a blank to indicate that the file is now open for updating.

Example 2: Opening a Selected File

Assume that you are viewing the library `!Users.Anderson.Documentation` and you want to edit the file `History_Log`, which is not yet displayed. One option is to display the file using `[Definition]` and then open it for editing using the steps in the previous example.

Alternatively, you can *select* the library entry for `History_Log` and then use `[Edit]` to both display and open the selected file, without using `[Definition]` as a separate step.

To both display and open the `History_Log` file for editing:

1. Start with the cursor in the library containing `History_Log`—namely, `!Users.Anderson.Documentation`.
2. Select the directory entry for `History_Log` by putting the cursor on it and then pressing `[Object] - [-]`. The entry is now highlighted.
3. Leave the cursor in the highlighted selection and press `[Edit]`. The `History_Log` file is displayed in a window with a blank banner symbol indicating that the file is open for updating.

Note that this use of `[Edit]` exemplifies a prevalent Environment usage pattern—namely, the pattern of first designating something to operate on and then operating on it. You have seen this pattern of usage with `[Definition]`, where cursor positioning designates what is to be displayed. `[Edit]` follows this pattern, except that selection must be used along with cursor positioning to designate what to open for editing. That is, selection serves as a more explicit form of designation than cursor positioning alone. Therefore, selection is generally required by operations like `[Edit]`, which

can potentially change objects, whereas cursor positioning is accepted by relatively harmless operations like `Definition`.

Selection is used in a number of ways throughout the Environment. Further uses of selection are covered in Chapter 8, "Modifying Text," and "Selecting Ada Constructs," in Chapter 11.

Write Locks

When you press `Edit` to open a file for editing, you are given a *write lock* on that file. As long as you have a write lock on a file, only you can update the file. Other users can view the file, but they cannot open it for editing.

The write lock on a file is retained until you explicitly release it. Any of the following operations release the write lock on a file (these are described in "Closing Files for Editing," below):

- Pressing `Promote`
- Pressing `Object` - `X`
- Pressing `Object` - `G`

If you do not release the write lock on a file before logging out, the write lock is automatically released when you log out.

You will get a message when you display a file that another user is editing. For example, if user Miyata is editing a file called `!Projects.Documentation.Bulletin_Board` and you display that file using `Definition`, the following message appears in your Message window:

```
Warning: !PROJECTS.DOCUMENTATION.BULLETIN_BOARD is currently open.
```

This message alerts you to the possibility that the file may change while you are looking at its image. You can periodically press `Object` - `L` to refresh the file's image.

If, at this point, you press `Edit` to open that file for editing, the file remains read-only and the following message appears in the Message window:

```
Unable to obtain file: LOCK_ERROR
```

If you urgently need to work on that file, you can find out who is editing it:

With the cursor anywhere in the window containing the file, press **What Locks**.

Information such as the following is displayed in an output window. This information includes the username and session of the user who currently has the write lock on the file:

```
!PROJECTS.DOCUMENTATION.BULLETIN_BOARD'V(8)
  Updater: Miyata.S_1 Job 224
```

Closing Files for Editing

Closing a file for editing releases your write lock on it and, if the file remains displayed, converts the image to read-only. Unless you want to make files available to other users, it is not imperative that you close files after you have finished with them; however, closing files when you no longer want to work on them is recommended as a good housekeeping practice.

You can close a file for editing using any of the following operations, depending on whether you want the file to remain displayed and whether you want unsaved changes to be discarded. With the cursor in the window containing the file:

- Press **Promote** to save changes to the file and leave the file displayed in its window. Note the equals sign (=) in the window banner, indicating that the file is now read-only.
- Press **Object** - **X** to save changes and remove the file's window.
- Press **Object** - **G** to discard changes and remove the file's window.

If you want to edit a file after it has been closed, you must use **Edit** to open it again and regain the write lock.

RATIONAL

Chapter 8. Modifying Text

The Environment provides standard text editing capabilities, including:

- Inserting new text
- Deleting, moving, copying, and transposing portions of text
- Retrieving deleted text
- Searching for and, optionally, replacing specified strings
- Adjusting character case within portions of text
- Centering, justifying, and filling text

These editing capabilities are available not only for editing text files, but for modifying other kinds of images as well. Operations for searching and copying (for “cut and paste”) are available in any window. Operations for inserting, deleting, rearranging, and searching for text are especially useful for editing Ada units and Command windows, in addition to text files. Operations for adjusting case and formatting are particularly useful for editing text files; these operations are less important for modifying Ada source code, since source code pretty-printing is performed by operations provided specifically for editing Ada units (see “Using the Format Key,” in Chapter 11). Note, however, that operations for controlling text format and case can be useful when modifying comments within Ada units.

Adding Text

By default, editing is done in *insert mode*, so that you can add or insert text by positioning the cursor at the desired location and then entering the desired characters. New characters are entered immediately to the left of the cursor. Subsequent characters on the same line are shifted to the right as necessary to make room for the new text.

Some users prefer to edit in *overwrite mode*, in which subsequent characters on the line are not shifted to the right but instead are replaced one for one by each new character entered.

You can change the mode of the particular file you are editing, without changing the default mode. To edit a particular file in overwrite mode:

1. Put the cursor anywhere in the window containing the file you are editing.
2. Press `Image` - `O`. The window banner displays the word OVERWRITE.

To return the file to insert mode:

1. Put the cursor anywhere in the window containing the file.
2. Press `Image` - `I`. The OVERWRITE indicator is removed from the window banner.

You can make overwrite mode the default by changing the value of the `Image_Insert_Mode` session switch; see the *Session and Job Management (SJM)* book of the *Rational Environment Reference Manual*.

Operating on Designated Text Items

Most of the editing operations covered in the following sections (for example, deleting, moving, and copying) act on specific structural components of text. The following sections describe the various kinds of text items to which such operations are sensitive and how you can designate each kind of text item.

Text Items

At a basic level, text is composed of *characters*, which are grouped into one or more *lines*.

Further groupings within text include:

- *Words*, which are contiguous sequences of characters delimited by blank spaces, most punctuation characters, and Ada delimiters. Word delimiters are defined by the `Word_Breaks` session switch.
- *Sentences*, which consist of one or more words delimited by either a blank line or a sentence delimiter. Sentence delimiters are any of the following characters: period (`.`), question mark (`?`), and exclamation mark (`!`). A sentence delimiter must be followed by two or more blank spaces.
- *Paragraphs*, which consist of one or more sentences delimited by blank lines.

In addition, you can define any arbitrary stretch of contiguous characters, possibly spanning multiple lines, up to the entire file.

Patterns in Editing Operations

The Environment provides specific operations for editing characters, words, lines, and larger regions such as sentences, paragraphs, and arbitrary stretches of text. These editing operations fall into several patterns, according to the kind of text item they affect and how you must designate the item to be affected:

- Operations that act on characters are typically bound to key combinations modified by `Control`. Such operations affect the character on which the cursor is located.
- Operations that act on words or lines are typically bound to item-operation key combinations containing `Word` or `Line`, respectively. Such operations affect the word or line on which the cursor is located.
- Operations that act on sentences, paragraphs, or arbitrary stretches of text involve two steps:
 1. Select the desired text item (see “Selecting Text Items,” below).
 2. Press the appropriate item-operation key combination. Key combinations that pertain to selected items typically contain `Region`.

Selecting Text Items

Selection is a means of designating an item for various Environment operations. Note that some editing operations—typically those that affect characters, words, and lines—do not require selection; use of cursor position alone is sufficient. However, other operations—typically those that affect sentences, paragraphs, or arbitrary stretches of text—require that those items be designated by selection. (Note that some operations—for example, moving—operate only on selected items, so even words and lines need to be selected for these operations.) Selecting an item causes that item to be displayed in a highlighted font; only one item on the screen can be selected at a time.

Selecting an Arbitrary Stretch of Text

To select an arbitrary stretch of text, you need to define the beginning and endpoints. You can use the following method to select any contiguous set of one or more characters, including whole or partial words, lines, and the like:

1. Move the cursor to the start of the text to be selected.
2. Define the starting point by pressing `Region` - `[]`.
3. Move the cursor to the end of the text to be selected.
4. Define the endpoint by pressing `Region` - `[]`. The selected stretch of text is highlighted.

Using Object Selection

You can always select a sentence or paragraph by defining its beginning and endpoints. Alternatively, you can take advantage of the editor's knowledge of objects like words, sentences, and paragraphs by using object selection operations, such as `Object - [-]` and `Object - [-]`. Object selection operations allow you to select such items without using the cursor to delimit them. (Note, however, that you can use object selection to select only a single sentence or paragraph at a time. If you want to select two or more consecutive items—for example, two paragraphs within a five-paragraph file—you must use `Region - []` and `Region - []` to define the beginning and endpoints of the two paragraphs.)

To select a sentence:

1. Put the cursor anywhere on the sentence to be selected.
2. Press `Object - [-]`. The word nearest the cursor is highlighted.
3. Press `Object - [-]` a second time. The sentence containing the highlighted word is selected.

Similarly, to select a paragraph:

1. Select a sentence within the desired paragraph.
2. Press `Object - [-]` again. The paragraph containing the highlighted sentence is selected.

You can use number keys on the numeric keypad as an alternative to repeatedly pressing `Object - [-]`. For example, you can select the sentence the cursor is in by pressing `numeric 2 - Object - [-]`. Similarly, you can select the paragraph the cursor is in by pressing `numeric 3 - Object - [-]`.

Selecting within a File's Structural Hierarchy

Successively pressing `Object - [-]` selects increasingly higher-level text items in the following structural hierarchy, starting with words:

- Word
- Sentence
- Paragraph
- Entire file

You can select successively smaller items in this hierarchy by pressing `Object - [-]`. For example, if you have selected a paragraph, and you decide you want to select just a sentence within that paragraph:

1. Position the cursor on the desired sentence within the selected paragraph.
2. Press `Object - [-]`.

You can select the next or previous item at the same level in this hierarchy by pressing `Object - []` or `Object - []`, respectively. For example, if you have selected a sentence, and you decide you want the next sentence selected instead:

1. Leave the cursor in the current selection.
2. Press `Object - []`.
3. At this point, you can press `Object - []` to return to the previous sentence.

Turning Selections Off

A selected text item in a file is automatically “unselected” if you:

- Select another item (in any window on the screen).
- Enter or otherwise modify text anywhere in the file.

You can also explicitly turn a selected item off and return it to the normal display font:

1. Put the cursor anywhere in the selection.
2. Press `Item Off`.

`Region - [X]` turns off a selection no matter where the cursor is, providing a somewhat more convenient alternative to `Item Off`:

Summary of Selection Operations

Table 8-1. Operations for Selecting Text

<i>Operation</i>	<i>Key Combination</i>
Select starting point of arbitrary stretch of text	<code>Region - []</code>
Select ending point of arbitrary stretch of text	<code>Region - []</code>
Select successively larger items	<code>Object - []</code>
Select successively smaller items	<code>Object - []</code>
Select next item at same level	<code>Object - []</code>
Select previous item at same level	<code>Object - []</code>
Turn off selection cursor is in	<code>Item Off</code>
Turn off selection, regardless of cursor position	<code>Region - [X]</code>

See also “Selecting Ada Constructs,” in Chapter 11.

Deleting Text

The Environment provides operations for deleting any of the text items listed in "Text Items," above. The most useful of these operations are listed in the following table. Note that some operations delete an entire item and other operations delete from the cursor to either the beginning or end of the designated item.

Table 8-2. Operations for Deleting Text

<i>Operation</i>	<i>Key Combination</i>
Delete the character under the cursor	Control - D
Delete the character to the left of the cursor	Delete
Delete the entire word containing the cursor	Word - D
Delete from the cursor to the end of the word	Word - K
Delete from the cursor to the beginning of the word	Word - Delete
Delete the entire line containing the cursor	Line - D
Delete from the cursor to the end of the line	Line - K
Delete from the cursor to the beginning of the line	Line - Delete
Delete the entire selected item	Region - D

Retrieving Deleted Text

Each time you delete a text item larger than a character, the deleted item is automatically pushed onto the *hold stack*, from which the item can be recovered. The hold stack can hold up to 100 items at a time; items are dropped from the bottom of the stack as necessary to make room for new ones.

Several operations automatically push items onto the hold stack besides deletion, including copying and moving operations (see "Copying and Moving Text," below). In addition, there is an operation for pushing a selected item onto the hold stack, without having to delete, copy, or move it first. This is useful when you want to insert the same item at various points in your text.

To retrieve text from the hold stack:

1. Press **Region** - **↑** to retrieve the item most recently pushed onto the stack. The item is inserted to the left of the cursor and is highlighted as a selection.
2. Press **Region** - **←** to retrieve items that were pushed onto the stack earlier. Successively pressing this key combination cycles through the stack. Each successively retrieved item replaces the highlighted item at the cursor. Stop when you find the item you want.

The following table summarizes hold stack operations:

Table 8-3. Operations for Retrieving Held Text

<i>Operation</i>	<i>Key Combination</i>
Retrieve most recently held item	<code>Region</code> - <code>↑</code>
Retrieve next held item	<code>Region</code> - <code>→</code>
Retrieve previous held item	<code>Region</code> - <code>←</code>
Push selected item onto hold stack	<code>Region</code> - <code>↓</code>

Copying and Moving Text

The Environment provides operations for copying or moving selected text items. Both copying and moving insert the selected text in the location indicated by the cursor's position. However, moving deletes the selected text from its original location, whereas copying leaves the selected text in place.

Note that you can copy or move text from one window to another, provided that the object in the "target" window has been opened for editing. (When you move text, the "source" object must also be opened for editing.) Moving or copying between windows is useful not only for transferring text between two text files, but also for:

- Copying material from a job output window into a log file
- Copying messages from the Message window into a file
- Copying Ada code from a Command window into an Ada unit
- Copying information from a library image into a file, and the like

Copying Text

To copy text:

1. Select the text to be copied. Note that the object containing the selected text can be read-only.
2. Move the cursor to the target location, which can be in the same window or in a different one.
3. Press `Region` - `C`. A copy of the selected text is inserted to the left of the cursor. The newly inserted copy remains highlighted for subsequent operations.

Duplicating a Line

A special-purpose copy operation exists for making a copy of a single line directly below the original line. To duplicate a line:

1. Designate the line by putting the cursor on it.
2. Press `Line` - `C`.

Moving Text

To move text:

1. Select the text to be moved. Note that the object containing the selected text must be open for editing.
2. Move the cursor to the target location, which can be in the same window or in a different one.
3. Press **Region** - **M**. The selected text is deleted from its original location and inserted to the left of the cursor. The newly inserted text remains highlighted for subsequent operations.

Summary of Copy and Move Operations

Table 8-4. Operations for Copying and Moving Text

<i>Operation</i>	<i>Key Combination</i>
Copy a selected item	Region - C
Move a selected item	Region - M
Duplicate a single line	Line - C

Transposing Text

The following table shows the Environment operations for reversing the order of two characters, two words, or two lines. The cursor's position designates the second of the two items to be transposed.

Table 8-5. Operations for Transposing Text

<i>Operation</i>	<i>Key Combination</i>
Transpose the current and previous characters	Control T
Transpose the current and previous words	Word - T
Transpose the current and previous line	Line - T

Searching and Replacing

The Environment provides operations for searching for strings and, if desired, replacing those strings. You can use search operations in any window—for example, in library displays, the Message window, a job output window, the Window Directory, and the like. The replace operations can be used only in images that can be updated—for example, files, Ada units, and Command windows.

The search and replace operations are directional. Forward searching finds the next occurrence of a string, starting from the cursor's position and continuing the search to the end of the image, if necessary. Reverse searching finds the previous occurrence of a string, starting from the cursor's position and continuing the search to the beginning of the image, if necessary. You can change the direction of the search at any point.

Search and replace operations are initiated when you press one of four key combinations:

Table 8-6. Search and Replace Key Combinations

<i>Direction</i>	<i>Search</i>	<i>Replace</i>
Forward	Control S	Meta S
Reverse	Control R	Meta R

When you start either a search or a replace operation, the Environment enters *composing mode* to allow you to compose the search string and, when relevant, the replacement string. Prompts for these strings appear in the Message window and the ...composing message appears in the Message window banner. While composing mode is in effect, any characters you enter appear at the appropriate prompt, although the cursor remains in the image you are searching.

Composing mode is terminated and the actual searching or replacing begins when you press one of the four key combinations listed in Table 8-6. The Environment remains in search/replace mode until:

- No more occurrences of the search string are found.
- You press any key—for example, to start making changes at the found/replaced string or to move the cursor.

When search/replace mode is terminated, the prompts are removed from the Message window.

You can resume a terminated search or replace operation by pressing the appropriate key combination from Table 8-6. The prompts are redisplayed, containing the strings from the previous operation. Since these are in prompt form, you can:

- Reuse the previous strings by pressing the search/replace key combination again.
- Enter new strings; the old ones disappear automatically.
- Turn a given prompt to text by pressing `Item Off` and then modifying the text.

Example: Searching for a String

To search forward from the cursor's position:

1. Start the search operation and enter composing mode by pressing `Control-S`. The SEARCH prompt appears in the Message window and the ...composing message appears in the Message window banner.
2. Enter the string you want to find. The string you enter appears at the SEARCH prompt in the Message window.
3. Leave composing mode and start the actual search by pressing `Control-S` again. The ...composing message is removed from the Message window banner. The cursor appears one character after the first occurrence of the string to be found.
4. At this point, you can:
 - Find each subsequent occurrence of the string by pressing `Control-S`.
 - Find a previous occurrence of the string by pressing `Control-R`.
 - Stop the search by pressing any key—for example, `↓`.

Summary of Search and Replace Operations

Table 8-7. Operations for Searching and Replacing

<i>Operation</i>	<i>Key Combination</i>
Start search operation, forward/reverse	Control S / Control R
Search for the next/previous occurrence of a string	Control S / Control R
Start search and replace operation, forward/reverse	Meta S / Meta R
Go from SEARCH prompt to REPLACE prompt	Next Item
Go from REPLACE prompt to SEARCH prompt	Previous Item
Complete composing, find next/previous occurrence	Meta S / Meta R
Replace current occurrence, find next/previous occurrence	Meta S / Meta R
Do not replace, find next/previous occurrence	Control S / Control R
Replace current occurrence, discontinue operation	numeric 0 - Meta S
Replace all occurrences, forward	numeric - 1 - Meta S
Replace all occurrences, reverse	numeric - 1 - Meta R
Abort from composing mode	Control G
Abort from searching (and replacing) mode	Any key—for example, f

Controlling Case and Text Format

The operations discussed below are useful primarily when you are editing documents that require simple text formatting. When you are editing Ada units, the character case of identifiers is automatically adjusted by the Environment's pretty-printing facility. (Note that some of the formatting operations may be of use when editing comments in Ada code.)

Changing Character Case

The Environment provides operations for changing the character case in the following ways:

- You can *capitalize* text, so that words start with uppercase characters; any non-initial uppercase characters are made lowercase.
- You can *uppercase* text, so that all characters are uppercase.
- You can *lowercase* text, so that all characters are lowercase.

The following table lists the operations for controlling case within various text items. Note that some of these operations affect only the text from the cursor position to the end of the designated item; other operations affect the entire item.

Table 8-8. Operations for Controlling Case

<i>Operation</i>	<i>Key Combination</i>
Capitalize the character under the cursor	Control ^
Capitalize from the cursor to the end of the designated word	Word . ^
Capitalize from the cursor to the end of the designated line	Line . ^
Capitalize the entire selected item	Region . ^
Uppercase* the character under the cursor	Control >
Uppercase from the cursor to the end of the designated word	Word . >
Uppercase from the cursor to the end of the designated line	Line . >
Uppercase the entire selected item	Region . >
Lowercase the character under the cursor	Control <
Lowercase from the cursor to the end of the designated word	Word . <
Lowercase from the cursor to the end of the designated line	Line . <
Lowercase the entire selected item	Region . <

* Uppercasing a character is equivalent to capitalizing it.

Adjusting Text Format

The Environment provides simple formatting operations for determining text width, producing filled or justified paragraphs, and centering lines of text. You can perform these formatting operations as you edit; the results are displayed in the window.

Setting Word Wrap for Text

Recall that images extend indefinitely to the right, beyond the window edge. By default, you can enter text indefinitely to the right on any line; the image is scrolled automatically as you continue to type. If you choose, you can use `[Return]` to start a new line at appropriate points—for example, if you want to keep lines within the width of the screen so that entire lines are visible at a glance.

Alternatively, you can set *fill mode* on to provide automatic word wrap, so that you can type continuously without using `[Return]` to keep text within the desired column width. When fill mode is on, the Environment fills successive lines as you enter text, automatically starting new lines when necessary to prevent text from extending beyond the specified *fill column*. New lines are started at the nearest word break. (Note that when fill mode is on, you can still use `[Return]` to start new lines as desired.)

Fill mode is off by default; you can set fill mode on for particular images without changing the default mode. When fill mode is on, the default fill column is 72 (see also “Changing the Fill Column,” below).

To edit with automatic word wrap, set fill mode on as follows:

1. Put the cursor anywhere in the window containing the image you are editing.
2. Press `[Image] - [F]`. The window banner displays the word FILL followed by a number indicating the fill column that is in effect for that image.

From now on, the text you enter in this window is automatically wrapped on or before the fill column.

When you no longer need automatic word wrap for the image you are editing, you can turn fill mode off as follows:

1. Put the cursor anywhere in the window containing the image.
2. Press `[Image] - [X]`. The FILL indicator is removed from the window banner.

You can change the fill mode default so that all images are edited with automatic word wrap by changing the value of the `Image_Fill_Mode` session switch; see the Session and Job Management (SJM) book of the *Rational Environment Reference Manual*.

Filling Existing Lines of Text

The fill operation allows you to fill text that has already been entered. Filling text adjusts the placement of words within a selected area so that each line contains as much text as it can without extending beyond the fill column.

You can use the fill operation whether or not fill mode is on. When fill mode is on, text is filled as you enter it; in contrast, the fill operation fills short lines and wraps long lines within existing text. If fill mode is already on, you probably will not need to use the fill operation unless you have modified the text after entering it.

To fill existing text:

1. Select the area within which text is to be filled (for example, `numeric 8` - `Object` - `[-]` selects a paragraph).
2. With the cursor in the selection, press `Region` - `Format`.

If you are editing comments in an Ada unit, you can request that the fill operation preserve (or insert) comment delimiters at the beginning of each line. See the `Editor.Region.Fill` command in the Editing Images (EI) book of the *Rational Environment Reference Manual*; see also "Entering Comments," in Chapter 11.

Justifying Text

The justify operation both fills and justifies text, adjusting word placement so that all lines extend exactly to the fill column. Blank spaces can be inserted between words to create an even right margin. You can use the justify operation whether or not fill mode is on. Note that you do not need to use the fill operation before justifying text.

To justify text:

1. Select the area within which text is to be justified (for example, `numeric 8` - `Object` - `[-]` selects a paragraph).
2. With the cursor in the selection, press `Region` - `Complete`.

If you are editing comments in an Ada unit, you can request that the justify operation:

- Insert comment delimiters at the beginning of each line
- Preserve or change the indentation level established by the first line in the selected area

For more information, see "Entering Comments," in Chapter 11.

Centering Lines

You can center individual lines of text within the established fill column as follows:

1. Put the cursor anywhere on the line to be centered.
2. Press `Line` - `g`.

Changing the Fill Column

The fill column determines the maximum line length allowed by:

- Automatic word wrap
- `Region` - `Format` operation
- `Region` - `Complete` operation

The fill column is also used to determine where to position a centered line.

By default, the fill column is 72. You can change the fill column for a particular image as follows:

1. With the cursor in the appropriate window, create a Command window.
2. Enter `set.fill_column` and press `Complete`.
3. At the prompt, enter the desired column number.
4. Press `Promote`.

You can change the default fill column for all images by changing the value of the `Image_Fill_Column` session switch; see the *Session and Job Management (SJM)* book of the *Rational Environment Reference Manual*.

Inserting Page Breaks

If you plan to print out your file, you can insert page breaks where you want the printing device to start a new page. The page break control character is `Control``L`.

To insert a page break:

1. Determine where to put the page break. The page break control character should be inserted at the beginning of a blank line between the last line of the current page and the first line of the new page.
2. Prepare to enter a control character by pressing `Control``'`.
3. Enter the page break control character by pressing `Control``L`.

The page break control character is displayed as a highlighted capital L in your file.

RATIONAL

Part III. Developing Simple Ada Programs

Contents

Chapter 9. Overview of Ada Unit Development	III-1
Ada Compilation Units	III-2
Ada Unit States	III-3
Source State	III-3
Installed State	III-3
Coded State	III-4
The Environment's Compilation System	III-5
A Sample Library	III-6
Example: Creating and Executing an Ada Procedure	III-7
Chapter 10. Creating, Saving, and Promoting Ada Units	III-13
Creating Ada Units	III-13
Determining Ada Unit Names and Subclasses	III-14
Creating Subprograms	III-15
Creating Package Specifications and Bodies	III-15
Saving Work in Progress	III-17
Discarding Changes Since the Last Save	III-17
Opening Existing Units for Editing	III-18
Write Locks	III-18
Versions of Units	III-19
Closing Source Units for Editing	III-20
Promoting Units to the Installed State	III-20
Installing Units with Dependencies	III-22
Reading the Compilation Log	III-24
Overview of Operations for Changing Unit State	III-25
Changing to a Relative State	III-25

Changing to a Specific State	III-25
Changing the State of a System of Units	III-26
Chapter 11. Using Ada-Specific Editing Operations	III-27
Using the Format Key	III-27
Example: Using Format to Enter a Function	III-28
Hints for Using the Format Key	III-32
Checking for Semantic Errors	III-36
Syntactic and Semantic Error Reporting	III-37
Selecting Ada Constructs	III-39
Kinds of Selection Operations	III-39
Selecting Larger or Smaller Ada Constructs	III-40
Selecting the Next or Previous Ada Constructs	III-42
Creating Private Parts	III-43
Creating Bodies	III-45
Entering Comments	III-47
Operations for Entering Comments	III-47
Inserting Page Breaks	III-48
Chapter 12. Executing and Testing Ada Programs	III-49
Promoting Units to the Coded State	III-49
Coding Individual Units	III-49
Coding Units with Dependencies	III-50
Executing Programs	III-51
Using a Command Window	III-51
Using Selection	III-52
Operations for Job Control	III-52
Common Errors	III-52
Testing Units and Systems	III-53
Saving Interactive Test Programs	III-55
Chapter 13. Debugging Ada Programs	III-57
Starting the Debugger	III-58
The Debugger Window	III-59
Controlling Program Execution	III-60
Automatic Source Display	III-60
Stepping Through a Program	III-61
Following the Program's Flow of Control	III-63
Stepping Over Subprogram Calls	III-66
Setting Breakpoints	III-67

Breakpoint Characteristics	III-68
Executing to a Breakpoint	III-69
Displaying Variable Values	III-70
Modifying Variable Values	III-71
Redisplaying the Current Location	III-71
Reexecuting a Program	III-71
Catching Exceptions	III-73
Examining the Stack of Subprogram Calls	III-74
Displaying the Call Stack	III-74
Displaying Qualified Names in the Stack	III-76
Traversing from the Call Stack	III-76
Displaying Parameter Values for a Frame	III-77
When You Have Finished Debugging	III-77
Chapter 14. Browsing Ada Programs	III-79
Where Is This Defined?	III-80
If Definition Fails	III-80
Example 1: Viewing the Definition of a Subprogram	III-81
Selection versus Cursor Position	III-83
Some Browsing Options	III-83
Example 2: Viewing the Definition of a Variable	III-83
Where Is This Used?	III-86
Example 1: Showing Variable References	III-87
Example 2: Showing Usages in Multiple Units	III-88
Chapter 15. Modifying Installed or Coded Programs	III-91
Elements That Can Be Changed Incrementally	III-92
If Dependencies Exist	III-92
Units and States	III-93
Using Incremental Operations	III-93
Incrementally Modifying an Element	III-94
Selecting One or More Elements	III-96
Using the Window Provided by an Incremental Operation	III-97
Incrementally Deleting an Element	III-97
Incrementally Adding an Element	III-99
Adding a New Declaration	III-99
Adding the Corresponding Body	III-100
Determining the Kind of Element That Is Added	III-103
Some Common Problems	III-105

Removing an Unwanted Prompt	III-105
Forgetting to Demote a Body	III-105
Selecting a Construct That Cannot Be Edited	III-106
Attempting to Change a Declaration That Has Dependents	III-106
Making Changes That Require Demotion	III-107

Chapter 9. Overview of Ada Unit Development

This and the following chapters in Part III cover the fundamental Environment facilities for developing Ada programs within a single library. This chapter describes many of the basic concepts that pertain to developing Ada compilation units in the Environment, including:

- The distinction between Ada units and text files in the Environment
- The *unit states* through which you can *promote* or *demote* Ada units as you develop them, establish dependencies among them, and prepare them for executing
- How Ada programs appear in Environment libraries

These concepts are then tied together in a short example showing the development process of a simple Ada procedure, from unit creation through execution.

“Creating, Saving, and Promoting Ada Units,” Chapter 10, provides more detailed information about creating new Ada units, opening existing units for editing, saving changes to units, and then promoting Ada units to the next state to facilitate parallel program development.

“Using Ada-Specific Editing Operations,” Chapter 11, describes how to take advantage of the Environment’s facilities for entering and modifying Ada code, including operations for providing syntactic completion and for checking syntactic and semantic consistency.

“Executing and Testing Ada Programs,” Chapter 12, describes how to prepare a program for execution and then execute it. In addition, suggestions are given for testing programs through Command windows.

“Debugging Ada Programs,” Chapter 13, describes how to use the Environment’s source-level Debugger to analyze and, if desired, modify the behavior of your program as it executes.

“Browsing Ada Programs,” Chapter 14, describes how to use the Environment’s interactive cross-referencing facilities.

“Modifying Installed or Coded Programs,” Chapter 15, describes Environment facilities for making incrementally compiled changes to installed or coded Ada programs.

Ada Compilation Units

Ada programs are built from Ada compilation units. According to the *Reference Manual for the Ada Programming Language*, Ada compilation units are:

- Procedure specifications and bodies
- Function specifications and bodies
- Package specifications and bodies
- Generic specifications and bodies
- Generic instantiations
- Subunits

Recall from Chapter 3, “Traversing the Rational Environment,” that Ada compilation units (Ada units) constitute a distinct class of Environment objects. That is, unlike other computer systems you may have used, the Environment does not store Ada units in files. Instead, the Environment stores Ada units in a structured representation to which information is added as units pass through various *unit states* (see below). This rich underlying representation (called DIANA) enables the Environment to provide:

- Editing operations that act on the specific structure of the Ada programs you write—for example, syntactic formatting and completion
- A compilation management system that automatically determines which units need to be compiled in a given Ada system and determines the correct compilation order
- A source-level debugger
- Facilities for browsing Ada programs—namely, finding where Ada identifiers are defined and where they are used

Ada Unit States

At any given time, an Ada unit is in one of four unit states:

- Source
- Installed
- Coded
- Archived

The first three states (source, installed, and coded) represent distinct phases in the development of Ada units. Much of Part III is devoted to describing when to change a unit from one state to the next during development and the consequences of putting units into each state.

The fourth state (archived) is available for the compact storage of units that are not of current interest. For example, deleted units are automatically placed in the archived state until they are expunged; see the Library Management (LM) book of the *Rational Environment Reference Manual*.

Source State

When created, Ada units begin in the source state. You will typically enter the bulk of a program's source code into units that are in the source state. You can also make arbitrary editing changes in a source unit using basic text editing operations (see Chapter 8, "Modifying Text") as well as more powerful Ada editing operations (see Chapter 11, "Using Ada-Specific Editing Operations").

An important feature of the source state is that you can use Ada editing operations to interactively detect and correct syntactic and semantic errors such as missing punctuation, misspelled keywords, and unresolved Ada identifiers. In fact, although you can save a syntactically incorrect or semantically inconsistent unit, you cannot advance (promote) such a unit to the next state (installed).

Installed State

When a unit is syntactically correct and semantically consistent, you can promote it to the installed state. At this point, the unit can be referenced from other units (for example, named in *with* clauses).

Installing a unit registers it in the library in which it was created. Once registered in a library, the unit's name can be resolved. Consequently, any units that *with* an installed unit can then be made semantically consistent and installed, if desired.

When dependencies exist among Ada units, the units must be installed in the order specified by Ada compilation rules. For example, a unit specification must be installed before its corresponding body.

Because the process of installing Ada units allows you to build systems of dependencies among these units, the following restrictions guarantee and preserve the integrity of Ada systems:

- Units can be put in the installed state only if they are syntactically and semantically correct.
- Units in the installed state cannot be modified arbitrarily using basic text editing operations (compare to the source state). Instead, installed units can be modified only by using *incremental operations*, which check for dependencies to prevent the invalidation of the Ada system.
- Units in the installed state can be demoted back to the source state if arbitrary changes must be made. However, to demote an installed unit that has dependents, you must demote its dependents as well.

Coded State

When you are ready to execute an Ada program or to unit-test some part of it, you can promote the relevant units to the coded state. Promoting units to the coded state causes object code to be associated with those units.

Like installed units, coded units are guaranteed to be syntactically correct and semantically consistent. Furthermore, coded units cannot be modified arbitrarily, although incremental operations can be used to make changes in certain coded units, provided that the changes do not invalidate any dependencies. Coded units can be demoted to either the source or the installed state, depending on the nature of the changes that need to be made.

Note that units can remain in the installed state until you are ready to execute your program. Keeping Ada units in the installed state allows you to build and verify a system of Ada dependencies without actually incurring the overhead of generating object code.

Alternatively, units can be promoted directly from the source to the coded state if no further development is needed before execution.

The Environment's Compilation System

In contrast to other computer systems you may have used, the basic way of compiling a program in the Environment is not done by invoking a batch-type compiler. Instead, as the previous sections suggest, compiling a program consists of promoting all of the program's units from the source state to the coded state, at which point the program can be executed. Thus, the logical phases of compilation are distributed across the various unit states. For example:

- Syntactic error checking (parsing) is done while units are in the source state.
- Semantic rules are verified and dependencies are set up in the installed state.
- Object code is generated only when a unit is promoted to the coded state.

By breaking up the compilation process across unit states, you do not incur the overhead for complete compilation until you are ready to execute a program. More specifically, you don't need to wait for object code to be generated to check for compiler-detected errors. The Environment also provides batch compilation facilities; see the Library Management (LM) book of the *Rational Environment Reference Manual*.

The Environment's compilation system thus is organized around a very different model than you may be used to. For example:

- On other computer systems you may have used, Ada units are represented as text files; batch tools (such as a compilers and linkers) transform these files into other files (such as object files, executable images). Thus a single executable Ada unit is represented as a series of files.
- In contrast, the Environment represents each Ada unit as a single object that passes through the phases of the compilation process as you change the unit's state. Multiple states replace the notion of multiple files.

Because each Ada unit is a single object in the Environment, you are ensured that the program you are executing matches the source code. That is, changing the program source code in a unit automatically destroys the object code as the unit changes state. On more conventional computer systems, changes made to source files may invalidate but do not destroy any corresponding object files. These remaining files can be mistaken for current files and executed.

A Sample Library

Figure 9-1 shows a sample library containing the units for a program that constructs and operates on complex numbers. In this program, the main procedure, `Display_Complex_Sums`, uses resources from packages `Complex` and `Complex_Uilities`.

```

!Users Anderson.Complex_Numbers : Library (World):
Complex           : C Ada (Pack_Spec);
Complex           : C Ada (Pack_Body);
Complex_Uilities  : C Ada (Pack_Spec);
Complex_Uilities  : I Ada (Pack_Body);
  .Image          : S Ada (Func_Body);
Display_Complex_Sums : C Ada (Proc_Spec);
Display_Complex_Sums : C Ada (Proc_Body);
List_Generic      : C Ada (Gen_Pack);
List_Generic      : C Ada (Pack_Body);
Sample_Input      : File;

```

```

← USERS ANDERSON-COMPLEX_NUMBERS |library| ... World

```

Figure 9-1. A Sample Library

The library entry for each unit shows, from left to right:

- The unit's name.
- The unit's state: `S` for source, `I` for installed, or `C` for coded. The library display is automatically updated whenever the unit state is changed.
- The unit's class (`Ada`) and subclass (for example, `Proc_Spec` and `Proc_Body` for procedure specification and body).

In this example, the function body `Image` is a subunit of the package body `Complex_Uilities`. `Image` is listed under `Complex_Uilities` and the name `Image` is preceded by a period to indicate that `Complex_Uilities.Image` is the qualified name of the subunit.

This example also shows that a library can contain other objects besides compiled units—for example, files such as `Sample_Input`.

Example: Creating and Executing an Ada Procedure

The following example shows how user Anderson creates and executes a short procedure, called `Display_Factorial`, in his home library. The specific operations presented in this example are discussed in detail in subsequent chapters.

1. Anderson first displays the library in which he wants to create the procedure. With the cursor in the library display, Anderson creates an empty Ada unit by pressing `[Create Ada]`.

```
Anonymous is created
= Rational (Delta) ANDERSON S_1
-----
!Users Anderson : Library (World):
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Complex_Numbers  : Library (World);
Documentation     : Library (Directory);
Factorial        : S Ada (Func_Spec);
Factorial        : S Ada (Func_Body);
Login            : C Ada (Proc_Spec);
Login            : C Ada (Proc_Body);
Memo_12_08_86   : File (Text);
My_Test_Data    : File (Binary);
S_1              : Session;
S_1_Switches    : File (Switch);
Tools           : Library (World);
= USERS ANDERSON (library) World
-----
[comp_unit]

-----
[COMP_UNIT] : 4 ada | Source
```

Figure 9-2. An Empty Ada Unit

As Figure 9-2 shows, a window is opened containing a prompt for a compilation unit, `[comp_unit]`. Note that the banner of this window also contains the compilation unit indication.

- Starting with the cursor on the [comp_unit] prompt, Anderson enters some text and then requests syntactic assistance by pressing **Format**. (Note that the prompt disappears when typed on.)

```
procedure display_factorial(n:natural)is
```

```
* [COMP_UNIT] (ada) Source
```

Figure 9-3. Before Pressing **Format**

- The text he gives indicates that the unit will be a procedure body. This is enough information for the format operation to supply other necessary keywords (such as begin and end Display_Factorial;) and to display a prompt for further information. Specifically, a [statement] prompt is displayed to indicate that one or more statements are required for minimal syntactic completeness. Note that the image is pretty-printed as well.

```
procedure Display_Factorial (N : Natural) is
begin
  [statement]
end Display_Factorial;
```

```
// [COMP_UNIT] (ada) Source
```

Figure 9-4. After Pressing **Format**

4. Putting the cursor on the [statement] prompt, Anderson enters part of the first statement and presses `Format` again. (Once again, the prompt disappears.)

```
procedure Display_Factorial (N : Natural) is
begin
  for i in 1..n
end Display_Factorial;
```

```
*-{COMP_UNIT}-{ada}-Source
```

Figure 9-5. Building the First Statement, Before Pressing `Format`

5. The format operation once again provides minimal syntactic completion and pretty-prints the results. Another [statement] prompt indicates the need for one or more statements to complete the for statement.

```
procedure Display_Factorial (N : Natural) is
begin
  for I in 1 .. N loop
    [statement]
  end loop;
end Display_Factorial;
```

```
#-{COMP_UNIT}-{ada}-Source
```

Figure 9-6. Building the First Statement, After Pressing `Format`

6. Anderson continues to add text, using basic text editing operations and periodically pressing **Format**, until the procedure seems reasonably complete. He then checks for semantic errors by pressing **Semanticize**.

Semantic errors found

```

Rational Delta: ANDERSON S_1
Users Anderson: Library (World):
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Complex_Numbers  : Library (World);
Documentation     : Library (Directory);
Factorial        : S Ada (Func_Spec);
Factorial        : S Ada (Func_Body);
Login            : C Ada (Proc_Spec);
Login            : C Ada (Proc_Body);
Memo_12_08_86   : File (Text);
My_Test_Data     : File (Binary);
S_1              : Session;
S_1_Switches    : File (Switch);
Tools            : Library (World);
_Ada_8_         : S Ada (Proc_Body);
Users Anderson: Library (World)
with Text_IO;
procedure Display_Factorial (N : Natural) is
begin
  for I in 1 .. N loop
    The_Result := The_Result * I;
  end loop;
  Text_IO.Put_Line (Natural'Image (N) & " ! " & Natural'Image (The_Result));
end Display_Factorial;
Users Anderson: ADA_8_ (V(1) (Ada) - Source

```

Figure 9-7. Checking for Semantic Errors

As Figure 9-7 shows, the semanticize operation:

- Finds and underlines several errors
 - Displays a message in the Message window
 - Displays the Environment's temporary name for the unit (_ADA_8_) in the window banner and in the library containing the unit
7. Anderson presses **Explain** to display further information about the errors in the Message window:

THE_RESULT denotes no defined object or value

8. To correct the errors, Anderson moves the cursor to the declarative portion of the procedure and enters the text of the required variable declaration:

```
the_result:natural:=1
```

He formats and semanticizes the unit again to pretty-print it and verify that there are no other errors.

9. Now that `Display_Factorial` is syntactically and semantically consistent, Anderson presses `Promote` to promote the unit to the installed state.

```
DISPLAY_FACTORIAL'Body changed to INSTALLED
= Rational (Delta: ANDERSON S_1
```

```
!Users Anderson : Library (World):
Calculation      : I Ada (Pack_Spec);
Calculation      : I Ada (Pack_Body);
Complex_Numbers  : Library (World);
Display_Factorial : I Ada (Proc_Spec);
Display_Factorial : I Ada (Proc_Body);
Documentation     : Library (Directory);
Factorial        : S Ada (Func_Spec);
Factorial        : S Ada (Func_Body);
Login            : C Ada (Proc_Spec);
Login            : C Ada (Proc_Body);
Memo_12_08_86    : File (Text);
My_Test_Data     : File (Binary);
S_1              : Session;
S_1_Switches     : File (Switch);
Tools            : Library (World);
```

```
=!USERS ANDERSON (Library) World
```

```
with Text_IO;
procedure Display_Factorial (N : Natural) is
  The_Result : Natural := 1;
begin
  for I in 1 .. N loop
    The_Result := The_Result * I;
  end loop;
  Text_IO.Put_Line (Natural'Image (N) & " ! " & Natural'Image (The_Result));
end Display_Factorial;
```

```
=!USERS ANDERSON: DISPLAY_FACTORIAL 'BODY' V1: (ada) Installed
```

Figure 9-8. After Installing the Unit

As Figure 9-8 shows, the word `Installed` appears in the unit's window banner and the procedure's actual name, `Display_Factorial`, replaces its temporary name in the library. For convenience, the specification for the procedure has been created and installed automatically.

10. To test `Display_Factorial`, Anderson presses `Promote` again to promote the unit to the coded state (for convenience, the procedure specification is coded automatically, too.) He then opens a Command window, enters the procedure name and an argument at the `[statement]` prompt, and presses `Promote` one more time to execute the procedure. The procedure's output is displayed in the Environment's standard output window.

```
with Text_IO;
procedure Display_Factorial (N : Natural) is
  The_Result : Natural := 1;
begin
  for I in 1 .. N loop
    The_Result := The_Result * I;
  end loop;
  Text_IO.Put_Line (Natural'Image (N) & " ! " & Natural'Image (The_Result));
end Display_Factorial;
```

```
Anderson: ANDERSON: DISPLAY_FACTORIAL BODY V111 .ada: Coded
```

```
Display_Factorial (5);
```

```
end;
```

```
5 ! 120
```

```
Anderson: BODY V111 & DISPLAY_FACTORIAL (Text)
```

Figure 9-9. Testing the `Display_Factorial` Procedure

Chapter 10. Creating, Saving, and Promoting Ada Units

This chapter describes how to:

- Create Ada units
- Save changes in Ada units
- Open existing units for editing
- Promote Ada units to the installed state so that other units can name them in *with* clauses

Operations for entering and modifying Ada code are described in Chapter 11, “Using Ada-Specific Editing Operations.”

Creating Ada Units

“Example: Creating and Executing an Ada Procedure,” in Chapter 9, illustrated the basic steps for creating an Ada unit. (These steps are summarized in this section.) You can use these steps to create an Ada unit of any kind. Further details about creating specific subclasses of units are described in “Creating Subprograms” and “Creating Package Specifications and Bodies,” below.

Note that each Ada compilation unit in your program must be created individually. This is necessary because each Ada unit in the Environment is a distinct object, whereas file-based development systems allow multiple units (for example, a specification and a body) to be put into a single file.

To create an Ada unit:

1. Display the library in which you want to create the new Ada unit. If you need to create a new library, see the *Rational Environment Basic Operations*.

2. With the cursor anywhere in the library, press `⌘ Create Ada`. As a result:
 - The following message appears in the Message window:


```
Anonymous is created
```
 - A window is created for the empty Ada unit. This window contains a `[comp_unit]` prompt indicating that a compilation unit is required. The indication `[COMP_UNIT]` appears in the window banner, as well.
 - The empty Ada unit is in the source state (indicated in the window banner) and is open for editing (the leftmost field of the window banner contains a blank).
3. Put the cursor on the `[comp_unit]` prompt and begin entering the desired Ada code. As you enter your program, bear in mind the following:
 - You can type indefinitely to the right or down, and you can use the basic text editing operations described in Chapters 7 and 8 to insert, delete, move, copy, search for, and replace text.
 - You do not normally need to capitalize characters, start new lines, indent lines, or enter cosmetic blank spaces, because pretty-printing is available using the Ada-specific format operation (see Chapter 11). The format operation also provides minimal syntactic completion and error checking.
 - Semantic error checking is available using the semanticize operation, and there are other Ada-specific operations for selecting Ada structures, entering comments, and so on (see Chapter 11).

Determining Ada Unit Names and Subclasses

The Ada code you enter determines an Ada unit's name and subclass (its identification as the specification or body of a subprogram, package, and the like). Each Ada unit starts out empty, or "anonymous." Then, when you enter a line such as the following, the unit's subclass is determined and can be listed in the library:

```
function Factorial (X : Natural) return Natural is
```

In this example, the unit is listed as a function body (`Func_Body`); note that without the reserved word `is`, the unit would be listed as a function specification (`Func_Spec`). You can change a unit's subclass at any time by changing the appropriate reserved words; the library listing is updated automatically.

A unit's name is also determined by its contents, although, unlike the subclass, the name does not appear in the library until you install the unit. Until then, the library lists the unit using a temporary name assigned by the Environment. For example, the library entry for the uninstalled unit mentioned above looks something like this:

```
_Ada_8_          : S Ada (Func_Body);
```

Once the unit is installed, its library entry contains the name `Factorial`:

```
Factorial       : I Ada (Func_Body);
```

Naming Ada units is thus very different from naming files. An Ada unit name is intrinsic to the unit itself and reflects its contents. In contrast, files are named as you create them and before you enter any text; the name is specified as part of the `Text.Create` command. A filename is thus a mnemonic tag that may but need not reflect the contents of the file.

Creating Subprograms

You can use the basic steps given above for creating both the specification and the body for library-level subprograms. However, because the specification for a subprogram repeats a portion of the subprogram body, the Environment can create the subprogram specification automatically. Accordingly, to create a function or procedure in a library:

Create the subprogram body, following the basic steps for creating an Ada unit and entering the appropriate reserved words.

When you install the subprogram body (see “Promoting Units to the Installed State,” below), a unit for the subprogram specification is created automatically.

Creating Package Specifications and Bodies

You can use the basic steps given above for creating both the specification and the body for packages. However, a more convenient alternative to creating a package body “by hand” is to use the Environment’s automated facility for generating a package body from its specification.

To create a package in a library:

1. Create the package specification first, following the basic steps for creating an Ada unit and entering the appropriate reserved words. (Note that you can use `Create Private` to generate templates for private types in package specifications; see “Creating Private Parts,” in Chapter 11.)

You can leave the package specification in the source state or promote it to installed.

2. With the cursor anywhere in the window containing the package specification, press `Create Body`. As a result:
 - The Environment uses the declarations in the package specification to generate a skeletal package body containing a template for each visible subprogram. The `[statement]` prompts indicate where statements need to be filled in.
 - The package body is displayed in its own window. The body is in the source state and is open for editing.

Figure 10-1 shows the skeletal package body that was generated from the specification for package `Complex`, which is displayed in the upper window.

```

package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
private
  type Number is
    record
      Real, Imag : Float;
    end record;
end Complex;

```

```

package body Complex is
  function Make (X, Y : Float) return Number is
  begin
    [statement]
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    [statement]
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    [statement]
  end Imaginary_Part;
  function Plus (X, Y : Number) return Number is
  begin
    [statement]
  end Plus;
end Complex;

```

Figure 10-1. After Using `Create Body` from the Package Specification for `Complex`

3. Complete the package body using basic text editing operations and the Ada-specific operations described in Chapter 11:
 - Enter one or more statements at each `[statement]` prompt. The prompts disappear as you type on them.
 - Add any desired nonvisible elements (that is, context clauses, subprograms, and the like, which are in the body but not in the specification).

Saving Work in Progress

As you develop Ada units in the source state, you can periodically save your editing changes by pressing `Enter`. You can use `Enter` to save units that contain errors or are incomplete.

To save changes in an Ada unit:

1. Put the cursor anywhere in the image of the Ada unit.
2. Press `Enter`.

The modification symbol in the banner changes from * or # to a blank, indicating that the changes have been saved and that the unit is still open for editing. You can continue entering or modifying Ada code. At this point, it is also safe for you to log out.

Use `Enter` to preserve work in progress while units are in the source state. (In fact, since you cannot open installed or coded units for editing, saving with `Enter` applies only to editing source state units.) Note that you can save units that contain prompts and error indications.

Promoting a unit to the installed or coded state implicitly saves changes, too (see “Promoting Units to the Installed State,” below). However, a source unit can be promoted only if it is semantically or syntactically consistent, so using `Enter` is the only way to save unfinished units that contain errors.

Pressing `Object` - `X` also saves changes, in addition to removing the window and closing the unit for editing.

Discarding Changes Since the Last Save

If you decide you do not want to save the most recent changes that you have made to a source unit, you can discard all unsaved changes by reverting the unit to the way it was the last time you saved it.

To discard unsaved changes and leave the unit open for further editing:

1. With the cursor anywhere in the unit, press `Object` - `L`.
A Command window is created containing the `Common.Revert` command.
2. Press `Promote` to execute the `Common.Revert` command.

Pressing `Object` - `G` also discards unsaved changes, in addition to removing the window and closing the unit for editing.

Opening Existing Units for Editing

When you create an Ada unit, it is automatically opened for editing, and you can continue to edit it as long as it remains open. However, if you want to modify an existing unit that is not already open, you must explicitly open it. (Of course, you can open a unit only if you have been granted write access to it.) If you try to type on an Ada unit that is not open, the following message is displayed in the Message window:

This image is read-only

As with opening text files, there are two alternatives for opening an Ada unit, depending on whether the unit is already displayed:

- If the desired Ada unit is already displayed, you can open it for editing as follows:
 1. Put the cursor anywhere in the window containing the unit.
 2. Press `[Edit]`. Note that the banner symbol changes from an equals sign (=) to a blank to indicate that the unit is now open for updating.
- If the desired Ada unit is not already displayed, you can both display the unit and open it for editing as follows:
 1. Start with the cursor in the library containing the desired unit.
 2. Select the library entry for the unit by putting the cursor on it and then pressing `[Object] - []`. The entry is now highlighted.
 3. Leave the cursor in the highlighted selection and press `[Edit]`. The selected unit is displayed in a window with a blank banner symbol indicating that the unit is open for updating.

You can open any unit that is in the source state. In addition, you can open an installed or coded unit for editing, provided that no other units depend on it. Note that opening an installed or coded unit automatically demotes that unit to the source state. If you attempt to open a unit that has dependencies, an *obsolescence menu* is displayed listing the dependent units that will have to be demoted to source as well. Extra steps are required to demote the desired unit and its dependents to source; alternatively, you may be able to modify the unit using incremental operations (see Chapter 15, “Modifying Installed or Coded Programs”).

Write Locks

When you press `[Edit]` to open an Ada unit for editing, you are given a *write lock* on that unit. As long as you have a write lock on a unit, no other user can view or edit that unit. Note that Ada units differ somewhat from text files in that a locked file can be viewed by other users (for example, using `[Definition]`), whereas a locked Ada unit cannot be viewed.

The write lock on a unit is retained until you explicitly release it. (See “Closing Source Units for Editing,” below.) If you do not release the write lock on a file before logging out, the write lock is automatically released when you log out.

You will get a message if you attempt to view or edit a unit for which another user has the write lock. For example, if user Anderson is editing a unit called !Users.Anderson.Complex_Numbers.Display_Complex_Sums and you try to display that unit using `[Definition]`, the following message appears in your Message window:

```
Operation failed - object is in use
```

If, alternatively, you use `[Edit]` to try to open that unit for editing, the following message appears in the Message window:

```
Object locked - DISPLAY_COMPLEX_SUMS currently in use
```

If you need to work on that unit, you can find out who is editing it:

1. Select the library entry for the unit by putting the cursor on it and then pressing `[Object]` - `[]`. The entry is now highlighted.
2. Leave the cursor in the highlighted selection and press `[What Locks]`.

Information like the following is displayed in an output window. The display lists entries for the selected unit and its image. Below each entry is the username and session of the user who currently has the write lock:

```
!USERS . ANDERSON . COMPLEX_NUMBERS . DISPLAY_COMPLEX_SUMS 'BODY' V(2)
    Updater: Anderson.S_1 Job 241
!USERS . ANDERSON . COMPLEX_NUMBERS . DISPLAY_COMPLEX_SUMS 'BODY' V(2) 'Image
    Updater: Anderson.S_1 Job 241
```

Versions of Units

Each time you open an Ada unit for editing, a new version of the unit is created. The unit's version number is displayed as a version attribute (for example, 'V(2)) following the unit name in the window banner. By default, the Environment retains one version in addition to the current version; previous versions are permanently deleted. You can increase the number of versions that are retained; see the Library Management (LM) book of the *Rational Environment Reference Manual*.

Ada units and text files differ with respect to when versions are created. Text file versions are created when files are saved, not when they are opened for editing. Ada unit versions are created when units are opened for editing, but not when they are saved.

Therefore, you can use `[Enter]` frequently to save changes in Ada units without creating a new version each time. This means that you can return the unit to either of two earlier stages in its development:

- You can revert the unit to the way it was the last time you saved it (see “Discarding Changes Since the Last Save,” above).
- You can restore the previous version of the unit, effectively discarding all changes (saved and unsaved) that were made since you opened the unit for editing (see “Undeleting Objects or Previous Versions in a Library,” in the *Rational Environment Basic Operations*).

Note that you can view retained versions of Ada units, although you cannot execute them unless they are restored. To view a previous version of an Ada unit, enter the Definition command, supplying the unit name with the appropriate version attribute. For example:

```
Definition ("Display_Complex_Sums'Body'V(1)");
```

Closing Source Units for Editing

When you no longer want to edit an Ada source unit, you can use any of the following operations to close the unit for editing. Closing a unit releases the write lock on it so that it is available for other users to view or edit. Note, however, that you do not need to close a unit in order to promote it or to log out.

With the cursor in the window containing the unit you want to close:

- Press - X, which additionally saves changes and removes the unit's window. Note that the unit can contain syntactic or semantic errors.
- Press - G, which additionally discards changes and removes the unit's window.
- Promote the unit to the installed or coded state, which can be done only if the unit is error-free.

If you want to edit a unit after it has been closed, you must use to reopen it and regain the write lock. Note that opening the unit again creates a new version.

For installed or coded units, - X and - G have the same effect—namely, removing a unit's window from the screen and from the Window Directory.

Promoting Units to the Installed State

When a unit is syntactically and semantically consistent and you are ready to integrate it with other units, you can promote the unit to the installed state. When a unit is installed, other units can name it in *with* clauses without incurring semantic errors. Promoting to installed also allows you to integrate a system of units without waiting for object code to be produced. When you are ready to actually test your program, see Chapter 12, "Executing and Testing Ada Programs."

To promote a single unit to the installed state:

1. Put the cursor in the window containing the unit you want to install. The unit can, but need not, be open for editing.
2. Press either or .

A message is displayed in the Message window indicating that the unit is changed to installed and the word `Installed` appears in the window banner.

When you install a unit for the first time, its Ada name (for example, Factorial) is listed in the enclosing library, replacing the temporary name that was assigned by the Environment (for example, `_Ada_8_`). The Ada name remains in the library even if you demote the unit to source again. Note that until you install the unit for the first time, you can change a unit's name arbitrarily; however, once the unit has been installed, you can change its name only by *withdrawing* the unit from the library, regardless of the current unit state. (See "Changing the Name or Kind of an Ada Unit," in the *Rational Environment Basic Operations*.)

Installing a unit has several other effects, depending on the unit:

- If the unit was open for editing, installing it automatically saves changes, if any, and releases the write lock. As a result, the image becomes read-only and the = symbol appears in the window banner.
- If the unit is a subprogram body, installing it automatically creates a separate specification. Note that this does not apply to packages.
- If the unit names other installed units in *with* clauses, those other units are locked so that they cannot be demoted to source without taking extra steps to demote the newly installed unit, too.

The install operation implicitly checks for syntactic and semantic errors. If errors are found, the unit cannot be installed, and a message like the following is displayed in the Message window:

```
Promote failed - Semantic errors found
```

If the install operation fails:

1. Press `Explain` to display further information about the error(s).
2. Correct the error(s), using `Format` and `Semanticize` to verify that all errors have been corrected. Use `Enter` to save changes until the unit is ready to be installed.
3. Press `Promote` to install the unit.

Although a unit must be free of errors before it can be installed, it need not be complete. That is, a unit can contain [statement] prompts such as those inserted by `Create Body` and `Format` operations. (A unit cannot be installed if it contains any other kind of prompt, however). Promoting a unit that contains [statement] prompts allows continued development against partially completed work. For example, if a package contains two subprograms, one complete and one containing a prompt, you can install and ultimately code the package to test the finished subprogram. (If you try to execute the subprogram that contains the prompt, the `Program_Error` exception is raised when the prompt is encountered.)

Installing Units with Dependencies

When dependencies exist among Ada units, the units must be installed in the order specified by Ada compilation rules. For example:

- A unit specification must be installed before its corresponding body.
- A unit specification must be installed before other units that refer to it.
- A subunit must be installed after its parent unit.

If you install units individually using `Promote` or `Install Unit`, you must follow the specified order. For convenience, however, if you install a unit body first, the Environment will automatically install the unit's specification. (In fact, for subprograms, the Environment both creates and installs the specification.)

As an alternative to installing units individually, you can use `Install (This World)` to install a designated system of dependent units. This operation automatically detects dependencies and uses them to determine which units to install and the order in which to install them. Note that you can install a system of units even if some of those units are already in the installed state.

Before pressing `Install (This World)`, you must designate the system of units you want to install. There are several ways to do this:

- You can designate all the units in a library by putting the cursor anywhere in the window containing that library.
- You can designate a specific unit and all the units on which it depends by selecting the library entry for that unit. Alternatively, if that unit is already displayed, you can put the cursor in its window.

For example, consider the units in the world `!Users.Anderson.Complex_Numbers`, shown in Figure 10-2:

```

!Users.Anderson.Complex_Numbers : Library (World):
Complex                          : S Ada (Pack_Spec);
Complex                          : S Ada (Pack_Body);
Complex_Utilities                : S Ada (Pack_Spec);
Complex_Utilities                : S Ada (Pack_Body);
  Image                          : S Ada (Func_Body);
Display_Complex_Sums             : S Ada (Proc_Spec);
Display_Complex_Sums             : S Ada (Proc_Body);
List_Generic                     : I Ada (Gen_Pack);
List_Generic                     : I Ada (Pack_Body);
Sample_Input                     : File;

```

```

!USERS ANDERSON.COMPLEX_NUMBERS (library) world

```

Figure 10-2. The World `!Users.Anderson.Complex_Numbers`

The units in Figure 10-2 constitute a single program with `Display_Complex_Sums` as the main procedure. The dependencies among the units in this program are represented by arrows in the diagram in Figure 10-3 (unit bodies are shaded):

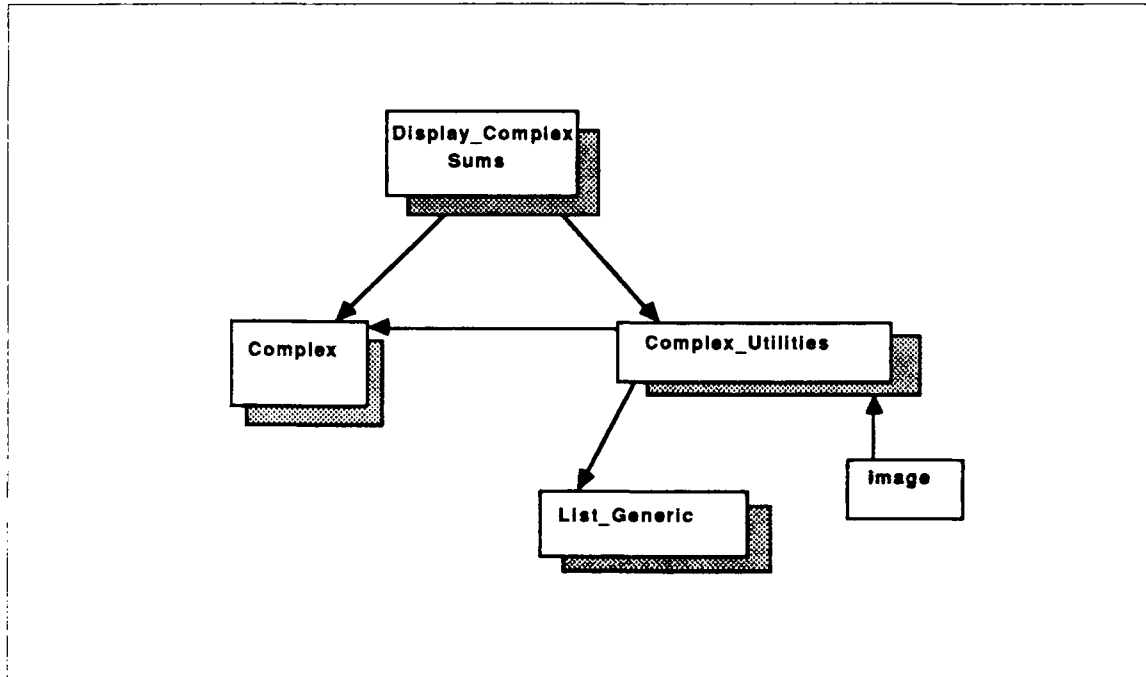


Figure 10-3. Dependencies among Units in `!Users.Anderson.Complex-Numbers`

To install all the units in the program represented in Figure 10-3:

1. Select the library entry for the the program's main procedure (`Display_Complex_Sums`) by putting the cursor on the entry and pressing `Object` - `[]`. You can select either the specification or the body. (As a shortcut, you can simply put the cursor in the library without selecting anything, since this library contains only units for this program.)
2. Press `Install (This World)`.


A log of messages indicating the operation's progress is displayed in an Environment output window. (A sample log is shown in "Reading the Compilation Log," below). In the library display, the symbol indicating installed state (I) replaces the symbol indicating source state (S).

Note that you can install a subset of a program by selecting a unit other than the main procedure. In the above example, selecting `Complex_Utilities` installs all units except `Display_Complex_Sums`.

Reading the Compilation Log

Figure 10-4 shows a sample compilation log for installing the units in world !Users.Anderson.Complex_Numbers.

```

-----
!USERS.ANDERSON.COMPLEX_NUMBERS % COMPILATION.PROMOTE  STARTED 1:29:28 PM
-----
87/07/30 13:29:29 ::: [Compilation.Promote ("", ALL_PARTS, INSTALLED,
87/07/30 13:29:29 ::: "<WORLDS>", FALSE, PERSEVERE).]
87/07/30 13:29:31 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS
87/07/30 13:29:31 ::: has been INSTALLED.
87/07/30 13:29:33 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX has been
87/07/30 13:29:33 ::: INSTALLED.
87/07/30 13:29:33 ::: !USERS.ANDERSON.COMPLEX_NUMBERS.LIST_GENERIC is already
87/07/30 13:29:33 ::: INSTALLED.
87/07/30 13:29:36 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES has
87/07/30 13:29:36 ::: been INSTALLED.
87/07/30 13:29:46 --- Messages generated while promoting !USERS.ANDERSON.
87/07/30 13:29:46 ::: COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS'BODY to INSTALLED.
87/07/30 13:29:46 >>> Procedure call TEXT_IO.PAT (" ");
87/07/30 13:29:46 *** Expanded name TEXT_IO.PAT is undefined.
87/07/30 13:29:48 +++ SEMANTIC_ERROR returned while attempting to promote
87/07/30 13:29:48 ::: !USERS.ANDERSON.COMPLEX_NUMBERS
87/07/30 13:29:48 ::: DISPLAY_COMPLEX_SUMS'BODY to INSTALLED.
87/07/30 13:29:49 +++ LOCK_ERROR returned while opening !USERS.ANDERSON.
87/07/30 13:29:49 ::: COMPLEX_NUMBERS.COMPLEX'BODY.
87/07/30 13:29:50 +++ LOCK_ERROR returned while attempting to promote !USERS
87/07/30 13:29:50 ::: ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY to INSTALLED.
87/07/30 13:29:50 ::: !USERS.ANDERSON.COMPLEX_NUMBERS.LIST_GENERIC'BODY is
87/07/30 13:29:50 ::: already INSTALLED.
87/07/30 13:29:59 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY
87/07/30 13:29:59 ::: has been INSTALLED.
87/07/30 13:30:02 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES.
87/07/30 13:30:02 ::: IMAGE'BODY has been INSTALLED.
87/07/30 13:30:03 ::: 2 units were already INSTALLED.
87/07/30 13:30:03 +++ 5 units were INSTALLED.
87/07/30 13:30:03 +++ 2 units could not be INSTALLED.
87/07/30 13:30:03 ::: [End of Compilation.Promote Command].

```

 COMPLEX_NUMBERS % COMPILATION.PROMOTE (Text)

Figure 10-4. Log Generated by Installing !Users.Anderson.Complex_Numbers

The log begins with a banner containing the world's name and the name of the command that is bound to `Install (This World)`. Each entry in the log shows a time stamp, a three-character symbol, and a message; the three-character symbols indicate information about the contents of the message (see below). The last four entries in this example summarize the results of installing the world !Users.Anderson.Complex_Numbers. Note from the summary that two units could not be installed because of errors reported earlier in the log.

The following three-character symbols characterize the messages in this log:

```

:::  Indicates current status information
+++  Indicates forward progress
...  Indicates message continuation
---  Indicates commentary
***  Indicates an error occurred
>>> Indicates the line containing the error
++*  Indicates a major step failed in the operation

```

In particular, note that the messages preceded by `++*` report that units could not be promoted for each of the following reasons:

- `Display_Complex_Sums'Body` contains a semantic error because a subprogram name was misspelled (and therefore could not be resolved).
- `Complex'Body` causes a lock error because another user had it open for editing when `Install (This World)` was pressed.

Note that if you display `Display_Complex_Sums'Body` at this point, the error that was reported is underlined. (See “Syntactic and Semantic Error Reporting,” in Chapter 11.)

Overview of Operations for Changing Unit State

`Promote`, `Install Unit`, and `Install (This World)` are part of a set of operations for promoting and demoting units and systems of units. Within this set are several groups:

- Operations that change a unit’s state relative to its current state.
- Operations that change a unit’s state to a specific state, regardless of its current state.
- Operations that change the state of a unit and all the units on which it depends. These operations invoke the Environment’s automated compilation facility to determine and use the correct compilation order.

Changing to a Relative State

Two operations change a unit’s state relative to its current state:

- `Promote` advances a unit from one state to the next—for example, from source to installed or from installed to coded.
- `Demote` changes a unit’s state in the reverse direction—namely, from coded to installed or from installed to source.

Changing to a Specific State

Three operations move a unit from any state directly to a specific state. Designate the unit by putting the cursor in the unit’s window or by selecting the unit’s entry in the library that contains it.

- `Source Unit` changes a unit's state from installed or coded directly to source (provided that there are no dependencies).
- `Install Unit` changes a unit's state from either source or coded directly to installed (provided that the unit contains no errors).
- `Code Unit` changes a unit's state from source or installed directly to coded (provided that the unit contains no errors).

Changing the State of a System of Units

A number of operations exist for changing the state of systems of units with dependencies. These operations differ with respect to:

- The direction of the change (whether units are promoted or demoted)
- The goal state (source, installed, or coded)
- The limit imposed on the change (whether units outside the current world can be affected)

These operations are summarized in Table 10-1:

Table 10-1. Changing the State of a System of Units

	<i>Source</i>	<i>Installed</i>	<i>Coded</i>
<i>Promote</i>		<code>Install (This World)</code> <code>Install (All Worlds)</code>	<code>Code (This World)</code> <code>Code (All Worlds)</code>
<i>Demote</i>	<code>Source (This World)</code> <code>Source (All Worlds)</code>	<code>Uncode (This World)</code> <code>Uncode (All Worlds)</code>	

In the key names, the limit of the change is indicated as follows:

- "This World" means that only units within the current world are changed, even if the system contains dependencies on units in other worlds.
- "All Worlds" means that units in any world can be changed, if required.

Thus, `Install (This World)` promotes units to the installed state, whereas `Uncode (This World)` demotes units to the installed state. Furthermore, `Install (This World)` looks only in the current world for units to install. If your program contains dependencies on units outside the current world, these units must already be installed or else you must use `Install (All Worlds)`.

Chapter 11. Using Ada-Specific Editing Operations

After you have created a new unit or opened an existing unit for editing, you can enter and modify Ada code. You can use any of the basic text editing operations (see Chapter 8, “Modifying Text”). In addition, you can use Ada-specific editing operations, including:

- `Format`, which provides syntactic completion, syntactic error checking, and pretty-printing.
- `Semanticize`, which provides semantic error checking.
- Operations for selecting structural components of Ada code, such as statements, parameter lists, expressions, identifiers, and the like. Once selected, these structures can be acted on by operations such as move, copy, and delete.
- Operations for entering comments and for commenting out portions of code.
- `Create Private` and `Create Body`, which generate templates for private types and unit bodies.

You are encouraged to make frequent use of Ada-specific editing operations. The first two operations listed above—`Format` and `Semanticize`—are particularly important because they enable you to detect and correct all syntactic and semantic errors during the process of entering code. This is essential in preparing a unit for promotion to the installed or coded state.

Using the Format Key

As you enter Ada code, you can use `Format` to help you enter the code faster, with less typing and fewer errors. `Format` does several important things in one step:

- `Format` provides minimal syntactic completion for incomplete Ada constructs, inserting:
 - Prompts for missing statements, expressions, identifiers, and the like
 - Ending punctuation such as right parentheses, closing quotation marks, and semicolons
 - Reserved words such as `begin`, `return`, `end if`;, and the like

- `Format` underlines any remaining uncorrected syntactic errors, such as out-of-context reserved words and punctuation. See “Syntactic and Semantic Error Reporting,” later in this chapter.
- `Format` pretty-prints the code to adjust the character case of identifiers, line breaks, indentation level, and spacing around certain delimiters and operators.

Therefore, when entering Ada code, you can enter short program fragments instead of complete Ada constructs, provided that you have given enough syntactic information to allow `Format` to fill in missing reserved words and punctuation. Furthermore, you do not need to enter code exactly as it should appear; rather, you can enter code in lowercase without line breaks or indentation.

`Format` is able to perform syntactic error checking, completion, and pretty-printing because it constructs an internal structured representation from the code you enter. Each time you press `Format`, the syntactic information in the underlying structure is updated to reflect any changes you made to the Ada code. The Ada unit you see in a window is actually the human-readable image of the underlying structure. Note that the underlying structure is the basis for other Ada-specific editing operations (such as selection), for browsing and cross-referencing facilities (see Chapter 15, “Browsing Ada Programs”), and the like.

Example: Using Format to Enter a Function

Assume that you have created an empty Ada unit and you want to begin entering a function. The following steps show how you might use `Format` to do this:

1. In the empty Ada unit, enter the Ada code fragment shown in Figure 11-1. This fragment contains enough information to be analyzed as a function named Example.

```
function example
```



Figure 11-1. Entering a Fragment for a Function

2. Press `Format` to format the fragment. As Figure 11-2 shows, `Format` supplies the reserved word `return`, a prompt for a required expression following the reserved word, and a final semicolon. You can fill in the prompt now or later.

Note that after you press `Format`, the # symbol replaces the * symbol in the window banner. (The * symbol indicates that the image has been changed; the # symbol indicates that the changed image has also been formatted.)

```
function Example return [expression].
```

Figure 11-2. After Pressing `Format`

- Now you want function Example to have a parameter called X. You can enter (X in the parameter list location, as shown in Figure 11-3:

```
function Example(X return [expression].
```

Figure 11-3. Adding a Parameter

- Press `Format` to format the fragment. As Figure 11-4 shows, `Format` supplies the punctuation for a single parameter specification and a prompt where you can fill in the type mark and any initializing expression.

```
function Example (X : [expression]) return [expression].
```

Figure 11-4. After Pressing `Format`

5. So far, minimal completion has caused this unit to be a function specification, since the previous fragments did not cue `Format` to complete the unit as a body. You can leave the unit as a function specification or you can change it to a function body by adding another syntactic cue, such as the reserved word `is`, as shown in Figure 11-5:

```
function Example (X : [expression]) return [expression] is;
```

```
# {COMP_UNIT} - ada - Source
```

Figure 11-5. Adding Another Syntactic Cue

6. Press `Format`. The appropriate reserved words and a `[statement]` prompt are added, as shown in Figure 11-6:

```
function Example (X : [expression]) return [expression] is
begin
  [statement]
end Example.
```

```
# {COMP_UNIT} - ada - Source
```

Figure 11-6. After Pressing `Format`

This example illustrates several important points:

- `Format` can introduce one or more prompts into the code. Bear in mind that:
 - These prompts behave just like prompts in Command windows. Prompts disappear when you type on them, and you can move between multiple prompts using `Next Item`, `Previous Item`, `Next Prompt`, or `Previous Prompt`.
 - Each prompt must be filled in to produce syntactically correct code. Depending on the requested item, a given prompt can be filled in with one or more items. For example, you can add one or more statements at the `[statement]` prompt.
 - You are not restricted to entering code only at prompts; you can enter code anywhere it makes sense. For example, you can add declarations before `begin`, although no prompts are supplied there (declarations are not required to make a syntactically valid function).

- `Format` provides syntactic assistance dynamically, based on code you have entered or modified, rather than by expanding text templates.

For example, to change the name of the function `Example` to `Format_Example`, you need to edit only the first occurrence of the name before pressing `Format`, as shown in Figure 11-7:

```
function format_Example (X : [expression]) return [expression] is
begin
  [statement]
end Example;
```

```
# {COMP_UNIT} (ada) Source
```

Figure 11-7. Changing the Unit's Name to `Format_Example`

Figure 11-8 shows that `Format` automatically changes the name in the function terminator (`end Format_Example;`) to match the name you edited.

```
function Format_Example (X : [expression]) return [expression] is
begin
  [statement]
end Format_Example;
```

```
# {COMP_UNIT} (ada) Source
```

Figure 11-8. After Pressing `Format`

- Finally, `Format` allows you to include as much or as little information in an Ada fragment as you want to. Figures 11-1 through 11-6 show that you can start by formatting a small fragment, then add more syntactic cues and format again, and so on. However, you can achieve the same results by starting with a more complete fragment, such as `function example(x is`. In fact, if you enter a syntactically complete Ada construct, `Format` just checks its syntax and pretty-prints it.

Hints for Using the Format Key

You are encouraged to experiment with `Format` to develop your own strategies for using it. The following are hints for successful use of `Format`.

Use `Format` frequently. Using `Format` frequently, pressing it after entering short program fragments or making small changes, yields better results than pressing it once after accumulating changes. Frequent formatting is faster, provides immediate feedback regarding errors, and prevents errors by supplying required syntactic elements.

Take advantage of `Format` for entering compound statements. `Format` is especially useful when entering *if*, *case*, and *loop* statements. For example, to enter a *case* statement, you can enter the reserved word *case*, as shown in Figure 11-9:

```
function Example (X : Natural) return Natural is
begin
  case
end Example;
```

```
/* (COMP_UNIT) : ada) : Source
```

Figure 11-9. Entering a Case Statement

After pressing `Format`, you are prompted for an expression and at least one alternative, as shown in Figure 11-10:

```
function Example (X : Natural) return Natural is
begin
  case expression is
    alternative
  end case;
end Example;
```

```
/* (COMP_UNIT) : ada) : Source
```

Figure 11-10. After Pressing `Format`

If you want several alternative clauses, you can enter several instances of the reserved word when in the alternative prompt, as shown in Figure 11-11:

```
function Example (X : Natural) return Natural is
begin
  case [expression] is
    when when when
  end case;
end Example;
```

```
# {COMP_UNIT} | ada | Source
```

Figure 11-11. Providing Cues for Three Alternative Clauses

Pressing **Format** provides syntax for alternative clauses that prompt for expressions and statements, as shown in Figure 11-12:

```
function Example (X : Natural) return Natural is
begin
  case [expression] is
    when [expression] =>
      [statement]
    when [expression] =>
      [statement]
    when [expression] =>
      [statement]
  end case;
end Example;
```

```
# {COMP_UNIT} | ada | Source
```

*Figure 11-12. After Pressing **Format***

Use a semicolon to terminate syntactic constructs. When entering a fragment among other Ada constructs, it is good practice to terminate the fragment you enter with a semicolon. The semicolon tells `Format` to ignore the subsequent constructs so that the new fragment can be completed as an independent statement or declaration.

Including a semicolon is especially important when entering:

- A compound statement among other existing statements
- A new subprogram among existing subprograms in a package

If you omit the semicolon, the newly completed fragment may incorporate subsequent constructs in an undesired way—for example, causing unwanted nesting of statements.

For example, assume that function `Example` contains the statement `Y := X + 1;` and you want to insert an `if` statement before it. To do this, enter the reserved words `if` then followed by a semicolon, as shown in Figure 11-13:

```
function Example (X : Natural) return Natural is
begin
  if then;
    Y := X + 1;
end Example.
```

```
## (COMP_UNIT) (ada) Source
```

Figure 11-13. Entering an If Statement

Pressing `Format` results in two separate statements as, shown in Figure 11-14 (note that the terminator `end if;` has been put before the assignment statement):

```
function Example (X : Natural) return Natural is
begin
  if [expression] then
    [statement]
  end if;
  Y := X + 1;
end Example;
```

```
## (COMP_UNIT) (ada) Source
```

Figure 11-14. After Pressing `Format`

If you had omitted the semicolon in Figure 11-13, pressing **Format** would have the effect shown in Figure 11-15:

```
function Example (X : Natural) return Natural is
begin
  if [expression] then
    Y := X + 1;
  end if;
end Example;
```

(COMP-UNIT) (ada) Source

Figure 11-15. After Pressing **Format**

Some common problems. **Format** completes fragments by interpreting the syntactic cues you give it. Sometimes this produces unintended results.

For example:

- As shown above, if you omit a semicolon after a fragment, **Format** may result in unwanted nesting of statements or subprogram declarations.
- If you enter a statement in the declarative portion of a block (before the `begin`), **Format** formats the statement as a declaration.
- If you misspell a reserved word, **Format** may interpret it as an identifier and format it as a procedure call or variable declaration, depending on its context.

To correct unintended formatting, delete the unwanted lines and enter a more complete fragment in the correct location.

Checking for Semantic Errors

You can use `Semanticize` to find semantic errors. `Semanticize` should be used periodically, although typically not as often as `Format`. You need not wait until a unit is complete before semanticizing it.

To check a unit for semantic errors:

1. Put the cursor anywhere in the window containing the unit. The unit must be open for editing.
2. Press `Semanticize`.

`Semanticize` updates the unit's underlying structure with semantic information and verifies that this structure conforms to the semantic rules of the Ada language. For example, `Semanticize` checks whether:

- Type compatibility is preserved among variables, expressions, and the like.
- Subprograms are used with the correct parameter profiles.
- Named objects have been declared. (`Semanticize` even checks the structures of units referenced in *with* clauses to make sure all Ada names can be resolved.)

The Message window displays a message indicating whether errors were found. (See "Syntactic and Semantic Error Reporting," below.)

`Semanticize` has several other effects:

- `Semanticize` saves changes, so that a blank replaces # or * in the window banner.
- When you press `Semanticize` for the first time after creating a unit, the unit's temporary name appears in the window banner.
- If you semanticize an unformatted unit, `Semanticize` performs an implicit `Format` operation and reports syntactic errors first, if there are any. No semantic errors are reported until you correct existing syntactic errors.
- `Semanticize` makes it possible to use `Definition` to browse through a unit (see Chapter 14, "Browsing Ada Programs").

Note that promoting a unit to installed implicitly semanticizes it, adding semantic information to the unit's underlying representation and reporting any errors. Therefore, you can press `Semanticize` before installing a unit, but it is not required.

Syntactic and Semantic Error Reporting

If one or more errors are found by `Format`, `Semanticize`, or an operation that installs a unit, then:

- A message is displayed in the Message window, indicating whether syntactic or semantic errors have been found.
- The errors in the unit are underlined.

For example, pressing `Semanticize` reports two semantic errors in the unit in Figure 11-16. These errors are underlined, and the cursor is positioned at the first error.

```
Semantic errors found
Rational (Delta) ANDERSON S_1
function Test (X : Integer) return Boolean is
  Answer : Boolean := False;
begin
  if Y > 0 then
    Answer := True;
  end if;
  return X;
end Example;
[COMP_UNIT] (ada) Source
```

Figure 11-16. After Pressing `Semanticize`

You can get more information about an underlined error by putting (or leaving) the cursor on the error and pressing `Explain`. As shown in Figure 11-17, the detailed error message appears in the Message window:

```
Y denotes no defined object or value
Rational (Delta) ANDERSON S_1
function Test (X : Integer) return Boolean is
  Answer : Boolean := False;
begin
  if Y > 0 then
    Answer := True;
  end if;
  return X;
end Example;
[COMP_UNIT] (ada) Source
```

Figure 11-17. Explaining the First Error

You can correct the error immediately or fix it later. In this example, you make the correction by changing Y to X.

Now, you move the cursor to the next error. You can move the cursor directly from one error to another by using the following keys:

- `Next Item` or `Next Underline` moves the cursor to the next error. Note that `Next Item` moves the cursor to prompts as well as errors, whereas `Next Underline` skips prompts.
- `Previous Item` or `Previous Underline` moves the cursor to the previous error. Note that `Previous Item` moves the cursor to prompts as well as errors, whereas `Previous Underline` skips prompts.

With the cursor on the second error, you can press `Explain` again to obtain more information, as shown in Figure 11-18:

```

INTEGER parameter X is not a value of type BOOLEAN
-----
--(Rational) (Data) ANDERSON $ _1
-----
function Test (X : Integer) return Boolean is
  Answer : Boolean := False;
begin
  if _x_ > 0 then
    Answer := True;
  end if;
  return X;
end Example;
-----
--(COMP_UNITS) (Code) $ source
-----

```

Figure 11-18. Explaining the Second Error

Note that, depending on the nature of a correction, some or all of the underline may remain until the next time you check for errors. You can remove all underlines by pressing `Underlines Off`.

Errors need not be corrected immediately. However, units containing errors cannot be installed. Until the errors in a unit are corrected, you can save the unit by pressing `Enter`.

Selecting Ada Constructs

As with text files, you can select portions of Ada units and then operate on the highlighted selections. For example, you can delete a selection with `Region - D`, copy it with `Region - C`, and so on. However, selection in Ada units is important for other reasons besides editing in the source state:

- Selection can be used in units of any state for browsing (see Chapter 14, “Browsing Ada Programs”).
- Selection is used in installed or coded units for performing incremental operations (see Chapter 15, “Modifying Installed or Coded Systems”).
- Selection is necessary for effective use of the Rational Debugger (see Chapter 13, “Debugging Ada Programs”).

Kinds of Selection Operations

You can select portions of Ada units using the selection operations that were introduced for text files (see “Selecting Text Items,” in Chapter 8):

- You can use `Region - I` and `Region - J` at desired cursor positions to define the beginning and endpoint of a selection.
- You can use object selection operations (such as `Object - --` and `Object - --`) to select structures without having to move the cursor to delimit them.
- You can use `Item Off` to turn off a selection (be sure the selection contains the cursor).

Object selection operations are based on the structure of the object in which they are used. Therefore, in Ada units, object selection operations do not select text structures such as words, sentences, and paragraphs. Instead, these operations use the underlying representation of Ada units to select Ada constructs at the desired level of program structure. Such constructs include declarations, statements, parameters, expressions, identifiers, and the like. Note that Ada comments constitute elements of program structure, so object selection operations do not treat them as text, even though they may contain words, sentences, and paragraphs.

Because object selection operations use the underlying representation constructed by the `Format` operation, object selection will fail if you have made changes since the last time you pressed `Format` or `Semanticize`. If a message such as the following appears in the Message window, simply press `Format` and try selecting again:

```
Selection failed - image is unformatted
```

Selecting Larger or Smaller Ada Constructs

Two of the object selection operations—namely, `Object` - `←` and `Object` - `→`—allow you to select constructs at different levels of program structure. Taken relative to a given point in the program, this means selecting successively larger or smaller constructs. For example, you can use these operations to select an entire block statement, an expression or statement within it, expressions and identifiers within any of the statements, and even identifiers or operators within the expressions.

To select an Ada construct:

1. Put the cursor in or near the construct you want to select.
2. Press either `Object` - `←` or `Object` - `→` to select the smallest Ada construct enclosing the cursor. Depending on the cursor's exact location, the initial selection may be larger or smaller than the construct you want to select.
3. If necessary, adjust the initial selection as follows:
 - Press `Object` - `←` repeatedly to select successively larger constructs enclosing the current selection.
 - Press `Object` - `→` repeatedly to select successively smaller constructs within the current selection. When no smaller construct exists, the selection is turned off so that nothing is selected.
 - You can use number keys on the numeric keypad as an alternative to repeatedly pressing `Object` - `←` or `Object` - `→`.

For example, Figure 11-19 shows a fragment of an Ada program with the cursor on the line containing the reserved word `begin`:

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
    Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
    Baseball.Sum (Current_Player, Team_Sums);
    Baseball.Add (Current_Player, Team_Statistics);
  end;
end loop;
```

SYSTEM.BASEBALL_STATISTICS.BODY.V111 (ada) - Source

Figure 11-19. A Program Fragment with Nested Statements

Pressing **Object** - **[-]** selects the block statement within the loop statement, as shown in Figure 11-20, because the block statement is the smallest complete construct enclosing the cursor:

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
    Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
    Baseball.Sum (Current_Player, Team_Sums);
    Baseball.Add (Current_Player, Team_Statistics);
  end;
end loop;

```

_SYSTEM BASEBALL_STATISTICS BODY v11.1 ada | Source

Figure 11-20. After Making the Initial Selection

Pressing **Object** - **[-]** narrows down the selection to a smaller construct containing or following the cursor. In this case, the statement list within the block is selected, as shown in Figure 11-21 (note that the cursor has moved automatically):

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
    Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
    Baseball.Sum (Current_Player, Team_Sums);
    Baseball.Add (Current_Player, Team_Statistics);
  end;
end loop;

```

_SYSTEM BASEBALL_STATISTICS BODY v11.1 ada | Source

Figure 11-21. Selecting the Statement List

Pressing `Object` - `←` again selects the first statement within the statement list, as shown in Figure 11-22:

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
    Baseball.Sum (Current_Player, Team_Sums);
    Baseball.Add (Current_Player, Team_Statistics);
  end;
end loop;
SYSTEM.BASEBALL_STATISTICS BODY V11 (ada) Source

```

Figure 11-22. Selecting the First Statement in the List

You can continue pressing `Object` - `←` to select `Data_Inputter.Get_Record` and then `Data_Inputter`. Alternatively, you can press `Object` - `←` to “retrace your steps.” For example, pressing `numeric 3` - `Object` - `←` in Figure 11-22 enlarges the selection two levels to select the block statement again. Finally, pressing `Object` - `←` one more time selects the enclosing loop statement.

`Object` - `←` and `Object` - `←` can be used similarly in declarations. For example, if you select an entire variable declaration, you can use `Object` - `←` to narrow down the selection to just the variable name.

Selecting the Next or Previous Ada Constructs

After selecting a construct at some level of program structure, you can select another construct at the same level as follows:

1. Move the cursor to the desired construct.
2. Press `Object` - `←`.

The old selection is “unselected” and the new one is highlighted.

Alternatively, you can use `Object` - `↑` and `Object` - `↓` as shortcuts for selecting the next and previous constructs at the same level without moving the cursor.

For example, assume that the first statement in the statement list is selected and that the cursor is in the selection, as shown in Figure 11-22, above. Pressing `Object` - `I` selects the next statement in the list, as shown in Figure 11-23:

```

loop
  declare
    Current_Player : Baseball.Player_Statistics;
  begin
    Data_Inputter.Get_Record (Current_Player);
    Baseball.Percentage (Current_Player);
    Baseball.Sum (Current_Player, Team_Sums);
    Baseball.Add (Current_Player, Team_Statistics);
  end;
end loop;

```

```

# {COMP_UNIT} (ada) Source

```

Figure 11-23. Selecting the Next Statement

At this point, you can reselect the previous statement by leaving the cursor in the current selection and pressing `Object` - `T`.

Similarly, a variable name and its type are treated as constructs at the same level of structure. Thus, when a variable name is selected in a declaration, you can use `Object` - `I` to select the type mark and `Object` - `T` to return to the variable name.

Creating Private Parts

You can use `Create Private` to generate completions for private parts in source state package specifications.

For example, assume that the specification for package `Complex` contains a private type declaration, as shown in Figure 11-24:

```

package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
end Complex;

```

```

# {COMP_UNIT} (ada) Source

```

Figure 11-24. A Package Containing a Private Type Declaration

To create a private part for this type:

1. With the cursor anywhere in the window containing the package specification, press **Create Private**.

As a result, the Environment creates a completion for a private region at the end of the package specification, as shown in Figure 11-25. Note that, by default, the completion is for a derived type:

```
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
private
  type Number is new [expression];
end Complex.
```

COMPLEX_NUMBERS.ADA:1:121 (ada) Source

Figure 11-25. Creating a Private Part Completion

2. Fill in the [expression] prompt and make any other necessary changes to complete the private part. Note in Figure 11-26 that the reserved word `new` has been deleted because `Number` is not a derived type:

```
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
private
  type Number is
    record
      Real, Imag : Float;
    end record;
end Complex;
```

COMPLEX_NUMBERS.ADA:1:121 (ada) Source

Figure 11-26. Completing the Private Part

If a specification contains more than one private type declaration, pressing **Create Private** once creates a private part containing completions for all of them.

Creating Bodies

You can use **Create Body** to:

- Create a skeletal body for an entire unit specification. In this case, the created body is itself a unit. (For an example, see “Creating Package Specifications and Bodies,” in Chapter 10.)
- Create a skeletal body for a new addition to an existing unit specification. The new body is created within the existing unit body. (An example of this case follows.)

Assume that the specification and body for package `Complex` exist as source state units, as shown in Figure 11-27:

```
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
private
  type Number is
    record
      Real, Imag : Float;
    end record;
end Complex;
```

```
COMPLEX_NUMBERS COMPLEX.v12 (ada) Source
```

```
package body Complex is
  function Make (X, Y : Float) return Number is 1
  begin
    return (X, Y);
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    return X.Real;
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    return X.Imag;
  end Imaginary_Part;
  function Plus (X, Y : Number) return Number is
  begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
  end Plus;
end Complex;
```

```
COMPLEX_NUMBERS COMPLEX_BODY.v11 (ada) Source
```

Figure 11-27. An Existing Package Specification and Body

If you add a function called `Minus` to the package specification, you can create a body for the new function by using the following steps.

1. Select the function declaration within the package specification, as shown in Figure 11-28:

```

package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
private
  type Number is
    record
      Real, Imag : Float;
    end record;
end Complex;

```

COMPLEX_NUMBERS.COMPLEX.V(2).ada Source

Figure 11-28. Selecting the Declaration for the Minus Function

2. Press **Create Body**. As a result, a skeletal body for the Minus function is added at the end of the package body for Complex, as shown in Figure 11-29:

```

package body Complex is
  function Make (X, Y : Float) return Number is
  begin
    return (X, Y);
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    return X.Real;
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    return X.Imag;
  end Imaginary_Part;
  function Plus (X, Y : Number) return Number is
  begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
  end Plus;
  function Minus (X, Y : Number) return Number is
  begin
    [Statement]
  end Minus;
end Complex;

```

COMPLEX_NUMBERS.COMPLEX.BODY.V(1).ada Source

Figure 11-29. The New Skeletal Body for the Minus Function

Entering Comments

No special treatment is needed for entering Ada comments, which are delimited on the left by two or more hyphens (--) and on the right by the end of the line. You can enter two kinds of comments:

- A right-trailing comment is on the same line and to the right of a line of Ada code.
- A stand-alone comment is on a line by itself.

Format and **Semanticize** ignore both kinds of comments, although **Format** adjusts the indentation level of certain stand-alone comments. **Format** pretty-prints comments as follows:

- Right-trailing comments are left in place; the number of blank spaces you leave between the Ada code and the comment are preserved.
- Stand-alone comments that are aligned under a right-trailing comment are left in place to form a multiline comment block at the same indentation level as the initial right-trailing comment.
- Stand-alone comment lines that begin in column 1 are left in place.
- Any other stand-alone comment lines are automatically indented to the level used for statements or declarations in the same position. Pretty-printing indents comments by inserting blanks to the left of the comment character (--).

A block of stand-alone comments (contiguous lines containing only comments) is treated as a single Ada construct and can be selected as a block using object selection operations. You must use **Region** - **[** and **Region** - **]** to select individual lines within a comment block or to select a right-trailing comment line.

Operations for Entering Comments

The Environment provides several operations for automatically entering right-trailing comments and for commenting out portions of code.

To enter a right-trailing comment:

1. With the cursor anywhere on the line to contain the comment, press **Meta Tab**.
As a result, a blank is inserted to the right of any Ada code on that line, followed by the comment characters (--). The cursor is positioned after the comment characters.

To comment out a portion of code:

1. Select the code you want to comment out.
2. Press **Region** - **[** to insert the comment characters (--) at the beginning of each line in a selected portion of code.

To “uncomment” code that had been commented out:

1. Select the code you want to restore.
2. Press `Region` - `+` to remove the comment characters (--) from the beginning of each line in a selected portion of code.

Note that all text editing operations are available for editing comments. In particular, comment characters can be inserted or preserved by the operations that fill and justify text (see “Controlling Case and Text Format,” in Chapter 8).

Inserting Page Breaks

If you plan to get a printout of your Ada units, you can insert page breaks where you want the printing device to start a new page. The page break control character is `Control` `L`.

To insert a page break:

1. Determine where to put the page break. The page break control character in an Ada unit should be inserted at the end of the last line to appear on the current page.
2. Prepare to enter a control character by pressing `Control` `]`.
3. Enter the page break control character by pressing `Control` `L`.

The page break control character is displayed as a highlighted capital L in the unit.

Chapter 12. Executing and Testing Ada Programs

This chapter describes how to:

- Prepare a program for execution by promoting it to the coded state
- Execute a program
- Use Command windows to interactively test units and systems

Promoting Units to the Coded State

When you are ready to execute an Ada program or to unit-test some part of it, you need to promote the desired unit (for example, the main procedure) and any units on which it depends to the coded state. Promoting units to the coded state causes object code to be associated with the underlying representation of those units.

The operations for coding units are parallel to the operations for installing units. In particular, there are two main ways of coding units:

- You can code units individually using `[Code Unit]` or `[Promote]`. This method is most useful when you want to test a few units where dependencies are not involved.
- You can code a system of units with dependencies using the automated compilation facility `[Code (This World)]`. This method lets the Environment manage the compilation order.

For more information on these operations, see “Overview of Operations for Changing Unit State,” in Chapter 10.

Coding Individual Units

To promote an individual unit to the coded state:

1. Put the cursor in the window containing the unit you want to code. The unit can be in the source or installed state.
2. Press `[Code Unit]` to put the unit directly into the coded state. (Alternatively, if the unit is already in the installed state, you can code it by pressing `[Promote]`.)

A message is displayed in the Message window indicating that the unit is changed to coded, and the word Coded appears in the window banner.

If you promote a source unit directly to coded, the unit is implicitly installed as well. Therefore, coding a source unit has the same consequences as installing a unit. Furthermore, coding and installing can fail for the same reasons—for example, if the unit contains syntactic or semantic errors. (See “Promoting Units to the Installed State,” in Chapter 10).

Coding Units with Dependencies

To code a system of units in a library, use the automated compilation facility as follows:

1. Put the cursor anywhere in the library containing the units you want to code.
2. Press `[Code (This World)]`. The Environment determines the compilation order among the units and changes all the units in the library to coded.

A log of messages indicating the operation’s progress is displayed in an Environment output window.

To code a specific unit and all the units on which it depends, use the automated compilation facility as follows:

1. Select the library entry for the unit.
2. Press `[Code (This World)]`. The Environment determines the compilation order among the units and changes the unit and its closure to coded.

A log of messages indicating the operation’s progress is displayed in an Environment output window.

The log produced by `[Code (This World)]` is similar to the log produced by `[Install (This World)]`, particularly with respect to error reporting. (For more information, see “Reading the Compilation Log,” in Chapter 10).

Executing Programs

Because the Environment uses Ada as its command language, you can execute your programs the same way you execute Environment commands—namely, by promoting them. Promoting to execution is thus the final step in a series of promotions, as indicated by Figure 12-1:

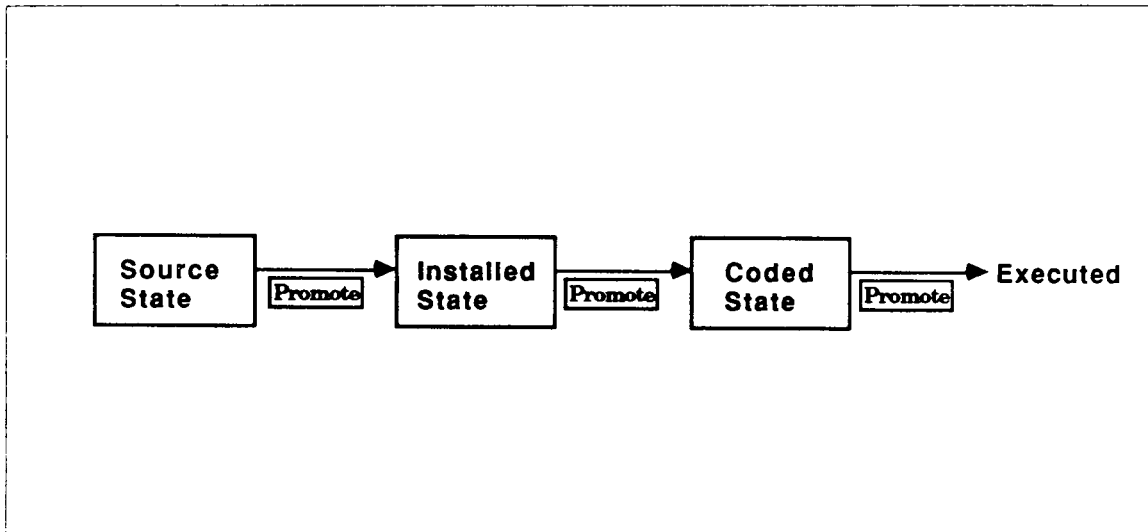


Figure 12-1. The Role of `Promote` from Source State to Execution

Using a Command Window

The basic way to execute a program is through a Command window:

1. Press `Create Command` to open a Command window from the library containing the program's main procedure.
2. At the `[statement]` prompt, enter the name of the program's main procedure with parameters as required (units must be in the coded state).
3. Press `Promote`.

Your program is linked dynamically to create an executable representation. The program is then elaborated and executed.

All of the operations and features of Command windows are available for entering and executing user-developed programs. For example, you can press `Complete` to get semantic completion for the procedure name and parameters.

Using Selection

When executing a coded procedure, you can use the following shortcut as an alternative to using a Command window:

1. Select the library entry for the procedure.
2. Press `Promote` to execute the procedure.
3. If the procedure requires parameters, a Command window is opened automatically, displaying the procedure name and parameter prompts. Fill in the prompts and press `Promote`.

Operations for Job Control

Following are useful operations for job control:

- To execute your program in the background as a batch job, press `Control Promote` instead of `Promote` in either of the methods described above.
- To interrupt your program, press `Control G`. Interrupting a job allows you to enter other commands while the interrupted job continues to run in the background.
- To kill your program before it completes, press `Job Kill`.

Common Errors

When you execute a program, you may encounter some common errors:

Not all units in the program are in the coded state. If you attempt to execute a program that contains one or more units that are in the source or installed state, a message like the following is displayed in the Message window:

```
1: ERROR !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY'V(5) is not coded
```

Note that such messages are prefixed with a number (for example, 1:) so that you can tell when multiple errors have been encountered.

To recover from this error, press `Code (This World)` to code the entire system of units.

An attempt was made to execute a subprogram that contains a [statement] prompt. It is possible to install and code units that contain incomplete subprograms (subprograms that contain [statement] prompts). This feature allows you to test completed subprograms in a package while other subprograms are still under development.

However, a program call to an incomplete subprogram raises the `Program_Error` exception, so that a message like the following is displayed in the Message window:

```
ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY'V(5)  NUM terminated due to unhandled
exception Program_Error (prompt executed)
```

To recover from this error, remove calls to the subprogram or complete its implementation.

Testing Units and Systems

Because you can enter and execute any Ada code in a Command window, you can use Command windows to build interactive unit and system tests quickly without having to build a separate, permanent program for every test. Testing through Command windows is useful when you need immediate feedback on specific aspects of your program's behavior. Of course, interactive testing is not intended to replace a permanent library of test programs. In fact, you can save interactive tests as permanent Ada units (see below).

The following example shows how you can use a Command window to unit-test the functions in a package. Assume that you have completed the implementation for most of the subprogram bodies in the package body !Users.Anderson.Complex_Numbers.Complex, shown in Figure 12-2:

```
package body Complex is
  function Make (X, Y : Float) return Number is
  begin
    return (X, Y);
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    return X.Real;
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    return X.Imag;
  end Imaginary_Part;
  function Plus (X, Y : Number) return Number is
  begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
  end Plus;
  function Minus (X, Y : Number) return Number is
  begin
    [statement]
  end Minus;
end Complex;
```

```
!Users.COMPLEX_NUMBERS.COMPLEX'BODY'V(1) (ada) Coded
```

Figure 12-2. The Package Body !Users.Anderson.Complex_Numbers.Complex

To exercise each completed function in package `Complex`, you can:

1. Create a Command window and enter a short program like the one in Figure 12-3. This test adds a complex number to itself and then invokes an Environment input/output operation (`Io.Put`) to display the results in a window.

```

package body Complex is
  function Make (X, Y : Float) return Number is
  begin
    return (X, Y);
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    return X.Real;
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    return X.Imag;
  end Imaginary_Part;
end Complex_Body;

```

~~COMPLEX NUMBERS~~ COMPLEX BODY V.1; (ada) Coded

```

declare
  use Editor, Ada, Common, Debug;
  Num : Complex.Number := Complex.Make (1.0, 2.0);
begin
  Num := Complex.Plus (Num, Num);
  Io.Put ("The real part is ");
  Io.Put (Complex.Real_Part (Num));
  Io.Put ("The imaginary part is ");
  Io.Put (Complex.Imag_Part (Num));
end;

```

Figure 12-3. A Sample Unit Test

Note that the following Command window features and Ada-specific editing features are available to help you enter test programs in Command windows:

- You can declare variables such as `Num` in the declarative portion of the Command window block statement.
 - You can use `Complete` to provide parameter prompts for each function. (However, to complete a function that occurs in the declarative region, such as `Complex.Make`, you must select the function name before pressing `Complete`.)
 - You can use `Format` to provide syntactic completion as you enter program fragments and to check for syntax errors.
 - You can use `Semanticize` to check for semantic errors.
2. Execute the test program by pressing `Promote`. (Because the test program is in a Command window, no state change is involved; one press of `Promote` is sufficient to execute it.)
 3. As usual, the statement portion of the test program turns into a prompt after you execute it. At this point, the following Command window features are helpful:

- You can modify and reexecute the test program, for example, to change the initial parameters for `Complex.Make`. Before modifying the statement portion of the program, however, press `Item Off` to turn off the prompt.
- You can use `Object - U` and `Object - R` to redisplay previously executed versions of the test program.

Saving Interactive Test Programs

Test programs written in Command windows are not permanent like Ada units created using `Create Ada`. That is, a test program in a Command window will not persist after you log out or use `Object - G` to remove the Command window or the window to which it is attached. If you want to save a test program that you wrote in a Command window:

1. Display the library in which you want to store the test program.
2. Press `Create Ada` to create an empty Ada unit.
3. Select the test program in the Command window.
4. Use `Region - C` to copy the selected program into the empty Ada unit.
5. Edit the program as necessary and save it using `Enter` or `Promote`.

RATIONAL

Chapter 13. Debugging Ada Programs

The Rational Debugger provides a variety of facilities for controlling and examining Ada programs during execution. The Debugger acts as an outside agent to a single executing program, allowing you to stop and restart it, step through it, display selected variable values, examine task state, and the like. Each interaction with the Debugger is reported in a special-purpose Debugger window, while the current location in the program source is automatically displayed in other Environment windows. Using these facilities, you can debug Ada programs at the Ada source level without having to know memory addresses or details of the underlying representations.

This chapter describes operations for:

- Starting, stopping, and restarting the Debugger
- Controlling program execution by stepping, setting breakpoints, and requesting continuous execution
- Displaying and modifying variable values
- Handling exceptions raised by the program
- Displaying and examining the stack of subprogram calls made by the program

The Debugger operations that are bound to function keys are found on the keyboard overlay under columns titled “Debug.”

This chapter covers only facilities for debugging programs that do not use tasking. In addition, the Debugger provides facilities for controlling and analyzing the individual tasks in programs that involve multiple tasks. For information on the Debugger and tasks, task state, and task control, see the Debugging (DEB) book of the *Rational Environment Reference Manual*.

Starting the Debugger

Starting the Debugger for a program is just like executing the program normally, except that you press `Meta Promote` rather than `Promote`. You do not need to specify special options in the program or to recompile it specially for debugging.

To start the Debugger for a particular program:

1. In a Command window, enter the main procedure (and parameters, if any) for the program you want to debug. All units in the program must be in the coded state.

As a convenient alternative to using a Command window, you can designate the program's main procedure by selecting its library entry.

2. Press `Meta Promote`.
3. After a brief pause, the Debugger window is displayed as shown in Figure 13-1, in the next section. You can now start and stop program execution using operations described in "Controlling Program Execution," below.

You can debug only one program at a time. If you start a second program under the Debugger before the current program has completed, the Debugger will kill the current program so that you can debug the next one. As a consequence, you can follow the steps above to restart a program or start a new program under the Debugger without waiting for the current program to complete.

Note that you can debug programs written in Command windows as well as in Ada units. For example, if you have used a Command window to write a test program, you can execute it under the Debugger by pressing `Meta Promote` as indicated above.

The Debugger Window

The Debugger window is displayed automatically when you start the Debugger. All interactions with the Debugger are displayed in this window as a sequential log. For example, assume that you have started executing the main procedure `Display_Complex_Sums` under the Debugger. As a result, the Debugger window is displayed, as shown in Figure 13-1:

```

!Users.Anderson.Complex_Numbers : Library (World):
Complex          : C Ada (Pack_Spec);
Complex          : C Ada (Pack_Body);
Complex_Uilities : C Ada (Pack_Spec);
Complex_Uilities : C Ada (Pack_Body);
Image           : C Ada (Func_Body);
=====
Display_Complex_Sums : C Ada (Proc_Spec);
Display_Complex_Sums : C Ada (Proc_Body);
List_Generic      : C Ada (Gen_Pack);
List_Generic      : C Ada (Pack_Body);
Sample_Input     : File (Text);
=====
!USERS.ANDERSON.COMPLEX_NUMBERS : Library (World)
Beginning to debug: !USERS.ANDERSON.COMPLEX_NUMBERS % !USERS.ANDERSON.
COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS
Stop at: .command_procedure, Root task: [Task : ROOT_TASK, #CCCC7].

=====
!R1000 Native Debugger

```

Figure 13-1. The Debugger Window

The Debugger window is identified by the word `(debugger)` in its window banner. The Debugger window banner also displays the words `R1000 Native`, which indicate that you are debugging a program executing “natively” on the R1000 Development System. (If your R1000 has a Rational Cross-Development Facility, you can also start a host/target debugger to debug programs that are executing on another target processor.)

Inside the Debugger window, several messages already appear in the log:

- The first identifies the name of the program that you are running under the Debugger.
- The second indicates that the Debugger is stopped just before executing the implicit command procedure that will in turn execute your program. Note that, although this program does not explicitly use tasking, messages such as this one refer to a program’s main thread of control as the *root task*.

As you debug the program, a message is added to the log to record each Debugger operation you use. You can scroll through the log in the Debugger window to see previous interactions, and you can save the log into a file by using the `Text.Write_File` command.

Controlling Program Execution

After starting the Debugger, program execution is under your control. You can cause the program to make forward progress or you can cause program execution to stop and wait for you to request further progress. While a program is stopped, you can use Debugger operations to display variable values, set breakpoints, examine the call stack, and the like.

There are several ways to make forward progress:

- You can run the program one step at a time, automatically stopping after each declaration is elaborated and each statement is executed. Stepping through a program allows you to analyze program behavior in great detail. The operations for stepping are `Run` and `Run (Local)`, described below.
- You can let the program execute continuously until some stopping point is reached—for example, an exception is raised, a breakpoint is encountered, or you stop the program with `Stop`. The operation for continuous execution is `Execute`.

There are several ways to cause continuous execution to stop:

- Before executing a given portion of the program, you can define a predetermined stopping place in that portion by setting a breakpoint. Program execution stops every time a breakpoint is encountered. (See “Setting Breakpoints,” below.)
- You can stop a program as it executes—for example, if you suspect the program is in an infinite loop. The operation for stopping execution is `Stop`.

A typical debugging session involves a combination of ways of controlling program execution. For example, you can set a breakpoint first and then let the program execute to the breakpoint, then single-step through various statements, then execute further until an exception is raised, and so on.

Automatic Source Display

When a program has stopped after making forward progress, the Debugger automatically displays the current location in the program source. More specifically, the Debugger automatically brings up the relevant Ada unit in a normal Environment window (similar to using `Definition`) and highlights the statement or declaration that will be executed or elaborated next. The current location is shown each time execution stops—for example, after a caught exception, a breakpoint, or a stepping operation.

If your program consists of multiple Ada units, the Debugger automatically displays the unit corresponding to the currently executing code. If the program calls a subprogram that is defined in another unit, the Debugger displays that unit in another Environment window. You can always use `Definition` to display other program units at any time during the debugging session.

Thus, when the program source is displayed, you can track your debugging progress in two ways:

- You can track the current location in the window containing the Ada unit.
- You can monitor the messages that are logged in the Debugger window.

Finally, if you debug Ada code that was composed in a Command window, this code is not automatically displayed in a separate window with the current location highlighted. However, if the code in the Command window code calls any subprograms that are defined in Ada units, the source for these subprograms is displayed.

Stepping Through a Program

The Debugger's stepping operations allow you to run a program one step at a time so that you can examine its behavior in detail. Following is the basic way to step through a program:

1. Press **Run**. The cursor can be anywhere on the screen.
As a result, a single declaration is elaborated or a single statement is executed. The next statement or declaration to be stepped through is automatically highlighted in the Ada unit.
2. If you want to progress faster than one step at a time, you can use keys on the numeric keypad to specify how many steps to elaborate or execute. Use **numeric 3** - **Run** to run three steps.

For example, after you start the Debugger for `Display_Complex_Sums`, the program is stopped, waiting for you to request forward progress (see Figure 13-1, above). Assume that you decide to step through the program to examine it in detail.

When you use stepping to start a program such as `Display_Complex_Sums`, you need to press **Run** twice to get into the program itself. Pressing **Run** the first time executes the implicit command procedure that calls `Display_Complex_Sums`. Pressing **Run** the second time actually executes the call to `Display_Complex_Sums`, at which point this unit is automatically displayed. The result of these first two steps is shown in Figure 13-2. Note that:

- Each **Run** operation has added a message to the Debugger window log.
- The first declaration in `Display_Complex_Sums` is highlighted, since that is the current location in the program's execution. This means that the next time you step, the highlighted declaration will be elaborated.

```

COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS
Stop at: .command_procedure, Root task: [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .command_procedure.1s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.1d [Task : ROOT_TASK, #CCCC7].

```

```

= F1000 Native (debugger)
with Complex, Complex_Uutilities, Text_io;
procedure Display_Complex_Sums is
package Complex_List renames Complex_Uutilities.Complex_List;
  List_Of_Numbers : Complex_List.List;
  Sum : Complex.Number;
  Input_Line : String (1 .. 100);
  Line_Length : Natural := 0;
  Input_File : Text_io.File_Type;
  Header : constant String :=
= .DISPLAY_COMPLEX_SUMS BODY v1.201 (ada) Coded

```

Figure 13-2. After Stepping Twice

Pressing **Run** a third time elaborates the highlighted declaration and causes the next declaration to be highlighted, as shown in Figure 13-3:

```

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .command_procedure.1s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.1d [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.2d [Task : ROOT_TASK, #CCCC7].

```

```

= R1000 Native (debugger)
with Complex, Complex_Uutilities, Text_io;
procedure Display_Complex_Sums is
  package Complex_List renames Complex_Uutilities.Complex_List;
List_Of_Numbers : Complex_List.List;
  Sum : Complex.Number;
  Input_Line : String (1 .. 100);
  Line_Length : Natural := 0;
  Input_File : Text_io.File_Type;
  Header : constant String :=
= .DISPLAY_COMPLEX_SUMS BODY v1.201 (ada) Coded

```

Figure 13-3. After Stepping a Third Time

Following the Program's Flow of Control

Stepping with **Run** allows you to follow a program's flow of control. That is, when executing a statement that contains a call to a subprogram, **Run** steps you through each declaration and statement in the called subprogram before returning to the portion of the program that contains the calling statement. Similarly, when elaborating a declaration that calls a function, **Run** steps through the function before returning to the calling declaration.

To illustrate how **Run** follows subprogram calls, assume that the current location is the highlighted statement shown in Figure 13-4. The corresponding message in the Debugger window indicates that this is statement 24 of `Display_Complex_Sums` (`.DISPLAY_COMPLEX_SUMS.24s`). This statement makes a call to the function `Complex.Plus`, and one of the parameters of `Complex.Plus` is itself a call to the function `Complex_List.Head`.

```
Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.23s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX_UTILITIES.COMPLEX_LIST.IS_NULL (new .LIST_GENERIC.IS_NULL).1s
      [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.24s [Task : ROOT_TASK, #CCCC7].
```

```
= R1000 Native debugger: current
-----
Text_io.Put_Line ("+" & Header & "+");
Text_io.Put_Line
  ("|          Value          Running Total          |");
Text_io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum = Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_io.Put ("| ");
  Text_io.Put (Complex.Utilities.Image
    (Complex_List.Head (List_Of_Numbers)));
-----
= .DISPLAY_COMPLEX_SUMS BODY v1201 ada: ~ Coded
```

Figure 13-4. Ready to Execute Statement 24 in `Display_Complex_Sums`

Since the most deeply nested subprogram must be executed first, pressing **Run** follows the function call to `Complex_List.Head`. Because `Complex_List` instantiates the generic package `List_Generic`, the source for `List_Generic` is automatically displayed, as shown in Figure 13-5. The first (and only) statement in the called function is highlighted, which is reflected in the corresponding message in the Debugger window.

```

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.23s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX_UTILITIES.COMPLEX_LIST.IS_NULL (new .LIST_GENERIC.IS_NULL).1s
      [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.24s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX_UTILITIES.COMPLEX_LIST.HEAD (new .LIST_GENERIC.HEAD).1s [Task :
      ROOT_TASK, #CCCC7].

```

```

= R1000 Native (debugger) current

```

```

function Head (Of_The_List : in List) return Element is
begin
return Of_The_List.all.Item;
end Head;

function Tail (Of_The_List : in List) return List is
begin
return Of_The_List.all.Next;
end Tail;

function Is_Null (The_List : in List) return Boolean is
begin

```

```

= : _NUMBERS LIST_GENERIC BODY (v1) (ada) Coded

```

Figure 13-5. Ready to Execute Statement 1 in `List_Generic.Head`

Pressing **Run** at this point executes the highlighted statement and follows the next subprogram call to `Complex.Plus`. As Figure 13-6 shows, the first (and only) statement in `Complex.Plus` is now highlighted:

```

    return (X, Y);
end Make;
function Real_Part (X : Number) return Float is
begin
    return X.Real;
end Real_Part;
function Imaginary_Part (X : Number) return Float is
begin
    return X.Imag;
end Imaginary_Part;
function Plus (X, Y : Number) return Number is
begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
end Plus;

end Complex;
# COMPLEX_NUMBERS.COMPLEX BODY V1.31 | ada | ~ Coded

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.24s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX_UTILITIES.COMPLEX_LIST.HEAD (new .LIST_GENERIC.HEAD).1s [Task :
    ROOT_TASK, #CCCC7]

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX.PLUS.1s [Task : ROOT_TASK, #CCCC7]

# R1000: Native debugger | current

function Head (Of_The_List : in List) return Element is
begin
    return Of_The_List.all.Item;
end Head;

function Tail (Of_The_List : in List) return List is
begin
    return Of_The_List.all.Next;
end Tail;

function Is_Null (The_List : in List) return Boolean is
begin
    return The_List.all.Is_Null;
end Is_Null;

# COMPLEX_NUMBERS.LIST_GENERIC BODY V1.1 | ada | ~ Coded

```

Figure 13-6. Ready to Execute Statement 1 in `Complex.Plus`

Finally, pressing **Run** once more returns to the original subprogram, `Display_Complex_Sums`. Now that statement 24 is completely executed, the next statement is highlighted (the Debugger refers to this statement as `.DISPLAY_COMPLEX_SUMS.25s`), as shown in Figure 13-7:

```

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX_UTILITIES.COMPLEX_LIST.IS_NULL (new .LIST_GENERIC.IS_NULL).1s
      [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.24s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX_UTILITIES.COMPLEX_LIST.HEAD (new .LIST_GENERIC.HEAD).1s [Task :
      ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX.PLUS.1s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.25s [Task : ROOT_TASK, #CCCC7].

= R1000 Native (debugger) current
-----
Text_io.Put_Line ("+" & Header & "+");
Text_io.Put_Line
  ("|          Value          Running Total          |");
Text_io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_io.Put ("| ");
  Text_io.Put (Complex.Utilities.Image
    (Complex_List.Head (List_Of_Numbers)));
= . . . DISPLAY_COMPLEX_SUMS BODY V1.20 | ada | Coded
-----

```

Figure 13-7. Ready to Execute Statement 25 in `Display_Complex_Sums`

Stepping Over Subprogram Calls

The Debugger provides a second stepping operation that allows you to “step over” calls to subprograms. This operation elaborates declarations and executes statements within each called subprogram without stopping after each individual step. Thus, you can step from one statement to the next within the same subprogram without stepping through any subprogram calls.

To step over subprogram calls:

Press **Run (Local)**. The cursor can be anywhere on the screen.

As a result, the program is executed or elaborated up to the next statement or declaration at the same level of program structure.

For example, using **Run (Local)** at statement 24 of `Display_Complex_Sums` allows you to step directly to statement 25 without the two intervening steps.

Setting Breakpoints

You can define predetermined stopping places in a program by setting breakpoints. Whenever a breakpoint is encountered, the program stops executing so that you can examine its behavior more closely at that point. For example, you can step through individual statements, examine variable values, and the like.

To set a breakpoint:

1. Display the Ada unit that contains the statement or declaration at which execution is to stop. If the unit has not been displayed already by the Debugger, use `Definition` or other traversal operations.
2. Find the last statement or declaration you want executed or elaborated before stopping. Use searching, scrolling, or stepping operations to find the desired program location.
3. Select the *next* statement or declaration. (If that statement or declaration is the highlighted current location, it is already selected.)
4. With the cursor in the selection, press `Break`. A breakpoint is set just before the selected location. This means that execution will continue up to but not including the selected location.

For example, assume that you selected and set a breakpoint at statement 25 in `Display_Complex_Sums`. A message reporting this is displayed in the Debugger window, as shown in Figure 13-8:

```
Run (STATEMENT, 1, "%ROOT_TASK");
Step: .COMPLEX.PLUS.1s [Task : ROOT_TASK, #CCCC7].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.25s [Task : ROOT_TASK, #CCCC7].

Break (".DISPLAY_COMPLEX_SUMS.25S", 1, "all", "TRUE");
The breakpoint has been created and activated:
Active Permanent Break 1 at .DISPLAY_COMPLEX_SUMS.25S [any task]
```

```
= R1000 Native (Debugger) current
-----
Text_Io.Put_Line ("+" & Header & "+");
Text_Io.Put_Line
  ("|          Value          Running Total          |");
Text_Io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_Io.Put ("|          |");
  Text_Io.Put (Complex.Utilities.Image
    (Complex_List.Head (List_Of_Numbers)));
= .DISPLAY_COMPLEX_SUMS BODY.v1201 (ada) Coded
```

Figure 13-8. Setting a Breakpoint at Statement 25 of `Display_Complex_Sums`

Breakpoint Characteristics

The message in Figure 13-8 reports that the breakpoint has been both created and *activated*. As long as a breakpoint is active, it will stop program execution. Breakpoints can be deactivated, for example, by using `Remove Breaks`. Inactive breakpoints have no effect on program execution, although information about them is remembered so that they can be reactivated using `Activate`.

Active breakpoints are deactivated automatically whenever you start a new program or restart the same program under the Debugger. If you restart the same program, you can press `Activate` to reactivate any existing breakpoints.

The message in Figure 13-8 also indicates that the breakpoint is *permanent*. A permanent breakpoint remains active as long as the program runs under the Debugger. A *temporary* breakpoint becomes inactive automatically after the first time it is encountered. Under the standard key bindings, `Break` sets permanent breaks and `Break Default` sets temporary breaks. (For more information on temporary breakpoints, see the Debugging (DEB) book of the *Rational Environment Reference Manual*.)

Finally, breakpoints are numbered, as shown in Figure 13-8, where the breakpoint is identified as Break 1. Various operations allow you to specify breakpoints by number. For example, you can deactivate a specific breakpoint by pressing its number on the numeric keypad and then pressing `Remove Breaks`. (With no numeric prefix, `Remove Breaks` deactivates all breakpoints.)

Press `Show Breaks` to display a list of all breakpoints, active or not, in the Debugger window.

Executing to a Breakpoint

Notice from Figure 13-8 that the breakpoint has been set within a loop statement. Therefore, pressing **Execute** causes the program to execute the rest of the loop, start the loop again, and stop just before statement 25.

As shown in Figure 13-9, statement 25 is highlighted and the last message in the Debugger window identifies the breakpoint that has been encountered:

```
Step: .COMPLEX.PLUS.1s [Task : ROOT_TASK, #CCCC7].
Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.25s [Task : ROOT_TASK, #CCCC7].
Break (".DISPLAY_COMPLEX_SUMS.25S", 1, "all", "TRUE");
The breakpoint has been created and activated:
Active Permanent Break 1 at .DISPLAY_COMPLEX_SUMS.25S [any task]
Execute ("all");
Break 1: .DISPLAY_COMPLEX_SUMS.25s [Task : ROOT_TASK, #CCCC7].
```

```

R1000 Native (debugger) --current
-----
Text_io.Put_Line ("+" & Header & "+");
Text_io.Put_Line
  ("|          Value          Running Total          |");
Text_io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_io.Put ("| ");
  Text_io.Put (Complex.Utilities.Image
    (Complex_List.Head (List_Of_Numbers)));
DISPLAY_COMPLEX_SUMS BODY Vt 20 | |ada| Coded

```

Figure 13-9. Executing to the Breakpoint

Displaying Variable Values

While execution is stopped, you can examine the values of variables to gather clues about program behavior up to this point. To display the current value of a variable:

1. Display an Ada unit containing an occurrence of the variable whose value you want to display.
2. Select the occurrence of the variable.
3. Press **Put**.

The variable's current value is displayed in the Debugger window.

For example, Figure 13-9, above, shows that `Display_Complex_Sums` has stopped just after executing statement 24, a statement that assigns a value to the variable `Sum`. You can see whether `Sum` has been assigned the correct value at this point by selecting `Sum` and pressing **Put**. Note that when you select `Sum`, the current location is "unselected" (it is no longer highlighted). Since `Sum` is a record, its values are displayed in Ada aggregate notation, as shown in Figure 13-10:

```
The breakpoint has been created and activated:
Active Permanent Break 1 at .DISPLAY_COMPLEX_SUMS.25S [any task]

Execute ("all");
Break 1: .DISPLAY_COMPLEX_SUMS.25s [Task : ROOT_TASK, #CCCC7].

Put ("%ROOT_TASK..1 SUM");

[ REAL => 9.699999999999999E+0000 [ 5460614548186726 * (2 ** -49)
  IMAG => 1.220000000000000E+0001 [ 6867989431740006 * (2 ** -49)
]
```

```
= R1000 Native (debugger) current
-----
Text_io.Put_Line ("+" & Header & "+");
Text_io.Put_Line
  ("|      Value      Running Total      |");
Text_io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_io.Put ("| ");
  Text_io.Put (Complex.Utilities.Image
    (Complex_List.Head (List_Of_Numbers)));
-----
= .DISPLAY_COMPLEX_SUMS.BODY.V120 (add) Coded
```

Figure 13-10. Displaying the Value of Sum

For more information on displaying portions of complex variables, see the Debugging (DEB) book of the *Rational Environment Reference Manual*.

Modifying Variable Values

Sometimes it is useful to analyze a program's behavior when its variables have different values. You can modify the values of scalar variables using `Modify`. To change the current value of a scalar variable to a desired value:

1. Locate and select an occurrence of the variable you want to modify.
2. Press `Modify` to open a Command window that contains the `Debug.Modify` command.
3. Enter the desired value at the `New_Value` prompt and press `Promote` to execute the command.

A message is displayed in the Debugger window reporting the variable's name, its old value, and its new value. The new value is used when you execute further.

To change the value of nonscalar variables, you must modify each of their scalar components individually. See the Debugging (DEB) book of the *Rational Environment Reference Manual*.

Redisplaying the Current Location

As you debug a program, you typically have occasion to scroll through the current program unit, view other program units, and use selection for a variety of Debugger operations, such as `Put`, `Modify`, and `Break`. Consequently, the unit containing the current location may no longer be displayed and, even if it is displayed, the specific statement or declaration may no longer be highlighted.

To return to the current location:

1. Press `Show Source`. The cursor can be anywhere on the screen, but not in a selection.
2. The Ada unit containing the current location is redisplayed if necessary, and the next declaration or statement to be elaborated or executed is highlighted. The current location is also reported in the Debugger window.

If you are debugging Ada code that was entered in a Command window, the current location in the code is displayed in the Debugger window.

If the Debugger window is replaced by subsequently displayed windows, you return to it by pressing `Debugger Window`.

Reexecuting a Program

In the course of analyzing a program's behavior, you may need to execute it under the Debugger more than once. This is the case if you have stepped or executed beyond a point of particular interest, if you have several sets of input to test, or if you have several different debugging strategies to try.

To restart the program under the Debugger, simply repeat what you did to start it:

- In a Command window, enter the name (and parameters) of the program's main procedure and press **Meta|Promote**.
- Alternatively, select the library entry for the program's main procedure and press **Meta|Promote**.

You do not need to wait for a program to finish executing under the Debugger in order to restart it. The unfinished job is killed automatically when the program is restarted.

For example, assume that you now want to execute `Display_Complex_Sums` under the Debugger with different input. To do this, you restart `Display_Complex_Sums` by selecting its library entry and pressing **Meta|Promote**, as shown in Figure 13-11. Messages in the Message window and in the Debugger window indicate that `Display_Complex_Sums` has been aborted and restarted.

```
!USERS.ANDERSON.COMPLEX_NUMBERS % !USERS.ANDERSON.COMPLEX_NUMBERS.DISPLAY_
COMPLEX_SUMS has been aborted
= Rational (Delta) ANDERSON S_1
```

```
!Users Anderson Complex Numbers : Library (World):
Complex : C Ada (Pack_Spec);
Complex : C Ada (Pack_Body);
Complex_Uilities : C Ada (Pack_Spec);
Complex_Uilities : C Ada (Pack_Body);
Image : C Ada (Func_Body);
Display_Complex_Sums : C Ada (Proc_Spec);
Display_Complex_Sums : C Ada (Proc_Body);
List_Generic : C Ada (Gen_Pack);
List_Generic : C Ada (Pack_Body);
Sample_Input : File (Text);
```

```
= !USERS.ANDERSON.COMPLEX_NUMBERS (library) world
```

```
Killing current job to begin debugging a new one.
-----
Beginning to debug: !USERS.ANDERSON.COMPLEX_NUMBERS % !USERS.ANDERSON.
COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS
Stop at: .command_procedure, Root task: [Task : ROOT_TASK, #59CE4].
```

```
= R1000 Native (debugger) current
```

Figure 13-11. Restarting `Display_Complex_Sums`

Note that any breakpoints you set still exist, but they are inactive. When you restart a program, you can reactivate the breakpoints by pressing **Activate**.

Catching Exceptions

Exceptions raised in a program often signal some event of interest to you during debugging. Therefore, the Debugger automatically stops program execution whenever exceptions are raised so that you can examine program behavior more closely at that point. This is called *catching* an exception.

For example, assume you have pressed **Execute** to run `Display_Complex_Sums`. As it executes, this program requests input of the form `<real_part, imaginary_part>`, and you enter incorrect input by omitting the angle brackets. As a result, the `Illegal_Complex_Number` exception is raised, execution stops, and a message reporting the exception is displayed in the Debugger window, as shown in Figure 13-12:

```

Enter source for complex numbers (t=terminal, f=file): t
Enter complex numbers in pairs of the form '<real_part, imaginary_part>',
terminate input with a line containing a single ':':
3.4,7.2

# ... COMPLEX_NUMBERS DISPLAY_COMPLEX_SUMS (TEXT) ... JOB 225 STARTED 2 07 15
Run (STATEMENT, 1, "%ROOT_TASK");
Step: .command_procedure.ls [Task : ROOT_TASK, #59CE4].

Run (STATEMENT, 1, "%ROOT_TASK");
Step: .DISPLAY_COMPLEX_SUMS.1d [Task : ROOT_TASK, #59CE4].

Execute ("all");
Exception: COMPLEX_UTILITIES.ILLEGAL_COMPLEX_NUMBER caught at
          COMPLEX_UTILITIES.VALUE.STRIP_LEADING_ANGLE.3s [Task : ROOT_TASK, #59CE4].

= R1000 Native (debugger) : current
procedure Strip_Leading_Angle (X : String;
                             Start_At : in out Scan_Range) is
begin
  Strip_Spaces (X, Start_At);
  if X (Start_At) /= '<' then
    raise Illegal_Complex_Number;
  end if;
  Start_At := Start_At + 1;
end Strip_Leading_Angle;

procedure Strip_Trailing_Angle (X : String;
                                Start_At : in out Scan_Range) is
begin
  Strip_Spaces (X, Start_At);
  if X (Start_At) /= '>' then
    raise Illegal_Complex_Number;
  end if;
  Start_At := Start_At + 1;
end Strip_Trailing_Angle;

# ... _NUMBERS COMPLEX_UTILITIES BODY (v15) (ada) ~ Coded

```

Figure 13-12. Catching the `Illegal_Complex_Number` Exception

If specific exceptions are not of interest to you during debugging, you can request that execution continue when they are raised. Consequently, these exceptions are

handled as usual by the program without being reported by the Debugger. Such exceptions are said to be *propagated* by the Debugger.

To request that the Debugger propagate a specific exception:

1. Display the Ada unit containing an instance of that exception—for example, the declaration for the exception or the exception handler.
2. Select the exception (or the statement containing it) and leave the cursor in the selection.
3. With the cursor in the selection, press **Propagate**. A message is displayed in the Debugger window. The next time that exception is raised, program execution continues with no special reporting by the Debugger.

To request that the Debugger resume catching a specific propagated exception:

1. Select the raise statement for the exception and leave the cursor in the selection.
2. Press **Catch**. A message is displayed in the Debugger window. The next time that exception is raised, program execution is stopped and the exception is reported by the Debugger.

Examining the Stack of Subprogram Calls

As a program executes, the Debugger maintains a record of the program's flow of control in the form of a *call stack*. Every time a subprogram is called, a *frame* for that call is pushed on the stack. The frame for a given subprogram call contains, among other things, a reference to the statement or declaration in the subprogram that made the call. The top, most recent frame in the stack always contains a reference to the current location in the program. Therefore, by examining the call stack for a program, you can reconstruct the sequence of calls that leads to the current location.

Examining a program's call stack can be especially useful when:

- The program raises an exception and stops automatically.
- The program has been executing an abnormally long time and you press **Stop** because you suspect an infinite loop.

In situations such as these, you can display the call stack to see where the program is and how it got there.

Displaying the Call Stack

To display a program's call stack:

1. If necessary, press **Stop** to stop program execution.
2. With the cursor anywhere on the screen, press **Stack**. The call stack is displayed in the Debugger window.

Note that when a program consists of a multiple tasks, each task maintains its own stack and these stacks can be displayed individually. For more information, see the Debugging (DEB) book of the *Rational Environment Reference Manual*.

For example, assume that `Display_Complex_Sums` has stopped executing because it raised the `Illegal_Complex_Number` exception. To see how the program got there, you press `[Stack]`, which displays the stack shown in Figure 13-13:

```

Execute ("all");
Exception .COMPLEX_UTILITIES.ILLEGAL_COMPLEX_NUMBER caught at
  COMPLEX_UTILITIES.VALUE.STRIP_LEADING_ANGLE.3s [Task : ROOT_TASK, #59CE4].

Stack ("XROOT_TASK", 0, 0);
Stack of task ROOT_TASK, #59CE4:
_1: STRIP_LEADING_ANGLE.3s
_2: VALUE.1s
_3: GET_LIST_OF_NUMBERS.4s.1s
_4: GET_LIST_OF_NUMBERS.4s
_5: DISPLAY_COMPLEX_SUMS.7s
_6: command_procedure.1s
_7: command_procedure [library elaboration block]

= R1000 Native (debugger) current
-----
procedure Strip_Leading_Angle (X : String;
                             Start_At : in out Scan_Range) is
begin
  Strip_Spaces (X, Start_At);
  if X (Start_At) /= '<' then
raise Illegal_Complex_Number;
  end if;

  Start_At := Start_At + 1;
end Strip_Leading_Angle;

procedure Strip_Trailing_Angle (X : String;
                               Start_At : in out Scan_Range) is
begin
  Strip_Spaces (X, Start_At);
  if X (Start_At) /= '>' then
raise Illegal_Complex_Number;
  end if;

  Start_At := Start_At - 1;
end Strip_Trailing_Angle;
-----
= .NUMBERS.COMPLEX_UTILITIES'BODY'v1.5:1:1: Code: Coded

```

Figure 13-13. Displaying the Call Stack for `Display_Complex_Sums`

The stack in Figure 13-13 contains seven frames. Each frame contains the location at which the program's flow of control changed. The topmost frame, frame `_1`, shows that the current location is statement 3 of the subprogram `Strip_Leading_Angle`. The next most recent frame, frame `_2`, indicates that `Strip_Leading_Angle` was called by statement 1 of the subprogram `Value`. Note that frame `_4` records the activation of a block—namely, statement 4 of the subprogram `Get_List_Of_Numbers`. The next subprogram call is in statement 1 of this block, as indicated by frame `_3`. The earliest frames, frames `_6` and `_7`, record the execution of `Display_Complex_Sums` itself.

Displaying Qualified Names in the Stack

The call stack displayed in Figure 13-13 refers to subprograms by their simple names. To request that the stack display qualified names:

1. In a Command window, enter the following command and press **Promote**:

```
Debug.Enable(Debug.Qualify_Stack_Names);
```
2. Redisplay the stack by pressing **Stack**. The stack with qualified subprogram names is shown in Figure 13-14:

```
_6: command_procedure.1s
_7: command_procedure [library elaboration block]

Enable (QUALIFY_STACK_NAMES, TRUE);
The QUALIFY_STACK_NAMES flag has been set to TRUE.

Stack ("%ROOT_TASK", 0, 0);
Stack of task ROOT_TASK, #59CE4:
_1: .COMPLEX_UTILITIES.VALUE.STRIP_LEADING_ANGLE.3s
_2: .COMPLEX_UTILITIES.VALUE.1s
_3: .DISPLAY_COMPLEX_SUMS.GET_LIST_OF_NUMBERS.4s.1s
_4: .DISPLAY_COMPLEX_SUMS.GET_LIST_OF_NUMBERS.4s
_5: .DISPLAY_COMPLEX_SUMS.7s
_6: .command_procedure.1s
_7: .command_procedure [library elaboration block]

= R1000 Native debugger: current
Debug.Enable (Debug.Qualify_Stack_Names
end;
```

Figure 13-14. The Call Stack with Qualified Names

Traversing from the Call Stack

You can use **Definition** to view any subprogram referenced in the stack. To do this:

1. Select the frame that contains the subprogram you want to view. Leave the cursor in the selection.
2. With the cursor in the selection, press **Definition**.

As a result, the Ada unit containing the subprogram is displayed. Within the subprogram, the statement or declaration containing the next subprogram call is highlighted.

Displaying Parameter Values for a Frame

It is often useful to know what parameter values were passed to a subprogram when it was called. You can use **[Put]** to display the parameter values that were passed to any subprogram referenced in the stack. To do this:

1. Select the frame that contains the desired subprogram. Leave the cursor in the selection.
2. With the cursor in the selection, press **[Put]**.

As a result, the Debugger window displays the values that were passed to the selected subprogram at the time it was called.

Each frame also contains the values of local variables as they were when the subprogram was called. For information about displaying variable values for a given frame, see the Debugging (DEB) book of the *Rational Environment Reference Manual*.

When You Have Finished Debugging

You are not required to take any special action to terminate the Debugger when you have finished debugging a program. The Debugger is terminated automatically if the program finishes executing or if you kill the program (for example, by pressing **[Job Kill]**). However, you can continue working in the Environment and even log out without taking extra steps to terminate the program you were debugging.

Note, however, that if your program is waiting for interactive input, the Quit command will display a warning. This is true whether or not a program is executed under the Debugger. At this point, you can kill the program or use **Quit(True)** to log out.

RATIONAL

Chapter 14. Browsing Ada Programs

The Environment provides facilities for *browsing* systems of dependent Ada units. Browsing a system of units allows you to look up cross-references interactively, without having to use printed cross-reference listings. For example, browsing is useful when you are:

- Analyzing someone else's program and you need to find type definitions for program variables
- Tracking down a problem with the Debugger and you need to see where and how a particular subprogram is implemented
- Considering a change to a subprogram and you want to know exactly where that subprogram is used

This chapter describes the Environment's facilities for browsing Ada programs, including:

- `Definition`, which answers the question: "Where and how is this object defined?"
- `Show Usage`, which answers the question: "Where and how is this object used?"

The examples in the following sections show how to browse portions of the Ada program `Display_Complex_Sums`. You may find it helpful to refer to Figure 10-3 in Chapter 10 to see how the units in this program are related.

Where Is This Defined?

In the process of analyzing, maintaining, or debugging a program, you typically need to know where and how various program elements are defined. With `Definition`, you can browse an Ada program to find this information. For example, you can use `Definition` to display the Ada unit in which:

- A given subprogram is declared or implemented
- A given variable or its type is declared

To view the defining occurrence of a program element such as a subprogram, a variable, or a type:

1. In an Ada unit, put the cursor on any occurrence of that element.

Alternatively, you can select any occurrence of the element. Selecting an element makes it easier to see what you've chosen. If you select the element, leave the cursor in the selection.

2. Press `Definition`.

As a result, the Ada unit containing the defining occurrence of the element is displayed.

(Note that `Definition In Place` has the same effect, except that the new window replaces the current one instead of being displayed in the least recently visited frame.)

By displaying various program units, `Definition` enables you to traverse an Ada program just as it enables you to traverse through the Environment library system, the list of windows in the Window Directory, the call stack for a program under the Debugger, and the like. Because `Definition` has such general usage, you are encouraged to try `Definition` whenever you want to traverse among related program elements.

If Definition Fails

You can use `Definition` in source, installed, or coded units. In a source state unit, `Definition` will fail with the following message if you made changes since you semanticized, demoted, or attempted to promote the unit:

```
Definition failed - not found
```

If this happens:

1. Press `Semanticize`.
2. Press `Definition` again. Note that `Definition` may fail if the cursor is on an underlined name (a name that cannot be resolved).

You can use `Definition` from a Command window only after executing the command or pressing `Semanticize` for an unexecuted command.

Example 1: Viewing the Definition of a Subprogram

Assume that you are viewing the body of the procedure `Display_Complex_Sums`, which contains a call to the function `Complex.Plus`. You want to know more about this function, so you select the function's identifier, as shown in Figure 14-1:

```

Text_Io.Put_Line
  ("|          Value          Running Total          |");
Text_Io.Put_Line ("| & Header & |");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_Io.Put ("| ");
  Text_Io.Put (Complex_Uilities.Image
    (Complex_List.Head (List_Of_Numbers)));
  Text_Io.Put (" ");
= DISPLAY_COMPLEX_SUMS BODY v1201 ada Coded

```

Figure 14-1. Selecting the Identifier `Complex.Plus`

Pressing `Definition` displays the specification of package `Complex`, which contains the declaration for `Plus`. Within the declaration, the identifier `Plus` is highlighted, as shown in Figure 14-2:

```

Text_Io.Put_Line
  ("|          Value          Running Total          |");
Text_Io.Put_Line ("| & Header & |");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_Io.Put ("| ");
  Text_Io.Put (Complex_Uilities.Image
    (Complex_List.Head (List_Of_Numbers)));
  Text_Io.Put (" ");
= DISPLAY_COMPLEX_SUMS BODY v1201 ada Coded

package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
private
  type Number is
= COMPLEX_NUMBERS COMPLEX v131 ada Coded

```

Figure 14-2. After Pressing `Definition`

For more information about Plus, you can press **Definition** again. This time, **Definition** displays the body of package Complex, which contains the subprogram body for Plus. Again, the identifier Plus is highlighted, as shown in Figure 14-3:

```

Text_Io.Put_Line
  ("|          Value          Running Total          |");
Text_Io.Put_Line ("| & Header & |");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_Io.Put ("| ");
  Text_Io.Put (Complex_Uilities.Image
    (Complex_List.Head (List_Of_Numbers)));
  Text_Io.Put (" |");
= ----- DISPLAY_COMPLEX_SUMS BODY v1.20 (ada) ----- Coded
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
private
  type Number is
    record
= ----- COMPLEX_NUMBERS_COMPLEX v1.31 (ada) ----- Coded
package body Complex is
  function Make (X, Y : Float) return Number is
  begin
    return (X, Y);
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    return X.Real;
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    return X.Imag;
  end Imaginary_Part;
  function Plus (X, Y : Number) return Number is
  begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
  end Plus;
= ----- COMPLEX_NUMBERS_COMPLEX_BODY v1.31 (ada) ----- Coded

```

Figure 14-3. After Pressing **Definition** Again

If you press **Definition** again, the cursor returns to the subprogram declaration in the package specification. Subsequent presses of **Definition** toggle the cursor between the specification and body for Plus. In this way, **Definition** and **Other Part** behave similarly.

Selection versus Cursor Position

Note that selection is used in this example to designate `Complex.Plus` because selection is the most visually explicit means of indicating what you want to see. Alternatively, you can use cursor position alone, although you must pay careful attention to where you put the cursor:

- Putting the cursor in the identifier `Plus` is equivalent to selecting the qualified name `Complex.Plus`, causing `Definition` to display package `Complex` and highlight the `Plus` function.
- Putting the cursor in the identifier `Complex` has a somewhat different effect, causing `Definition` to display package `Complex` without highlighting the `Plus` function.

Some Browsing Options

Under the standard key bindings, repeated uses of `Definition` display a unit's specification first, its body next, and then toggles between them. In the example above, `Plus` is found first in the specification of `Complex` and then in the body of `Complex`.

This sequence of events is controlled by the `Visible` parameter of the `Definition` command. Under the standard key bindings, `Visible` has the value `true`, so the specification (or visible part) is displayed first. To traverse directly to a unit's body, you can enter the `Definition` command with `Visible` set to `false`. (If this is your normal preference, you can rebind `Definition` to use this parameter value.)

Example 2: Viewing the Definition of a Variable

Assume that you have returned to the body of `Display_Complex_Sums` and you are looking once again at the assignment statement that contains `Complex.Plus`. Now you want to know more about the variable `Sum`, so you select the occurrence of `Sum`, as shown in Figure 14-4:

```

Text_io.Put_Line ("+" & Header & "+");
Text_io.Put_Line
  ("|          Value          Running Total          |");
Text_io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_io.Put ("| ");
  Text_io.Put (Complex_Uilities.Image
    (Complex_List.Head (List_Of_Numbers)));
  Text_io.Put (" ");
  Text_io.Put (Complex_Uilities.Image (Sum));
= DISPLAY_COMPLEX_SUMS BODY v1201 ada Coded

```

Figure 14-4. Selecting the Identifier `Sum`

Pressing **Definition** displays the declaration for Sum. Because this declaration is located earlier in the body of Display_Complex_Sums, the unit is scrolled in the same window. Within its declaration, the identifier Sum is highlighted, as shown in Figure 14-5:

```
with Complex, Complex_Uilities, Text_IO;
procedure Display_Complex_Sums is
  package Complex_List renames Complex_Uilities.Complex_List;
  List_Of_Numbers : Complex_List.List;
  Sum : Complex.Number;
  Input_Line : String (1 .. 100);
  Line_Length : Natural := 0;
  Input_File : Text_IO.File_Type;
  Header : constant String :=
    "-----";
= DISPLAY_COMPLEX_SUMS BODY v1201 (ada) Coded
```

Figure 14-5. Displaying the Declaration of Sum

Note that Sum is of Complex.Number type. Pressing **Definition** again displays the type declaration for Complex.Number, as shown in Figure 14-6:

```
with Complex, Complex_Uilities, Text_IO;
procedure Display_Complex_Sums is
  package Complex_List renames Complex_Uilities.Complex_List;
  List_Of_Numbers : Complex_List.List;
  Sum : Complex.Number;
  Input_Line : String (1 .. 100);
  Line_Length : Natural := 0;
  Input_File : Text_IO.File_Type;
  Header : constant String :=
    "-----";
= DISPLAY_COMPLEX_SUMS BODY v1201 (ada) Coded

package Complex is
  type Complex.Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
private
  type Number is
    record
= COMPLEX_NUMBERS COMPLEX v1131 (ada) Coded
```

Figure 14-6. Displaying the Declaration of Complex.Number

Because `Complex.Number` is a private type, pressing `Definition` one more time highlights the corresponding private part, as shown in Figure 14-7:

```

package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
private
  type Number is
    record
      Real, imag : Float;
    end record;
end Complex;

```

COMPLEX_NUMBERS COMPLEX.VI.31 | ada | Coded

Figure 14-7. Displaying the Private Part for `Complex.Number`

Subsequent presses of `Definition` toggle between the private type declaration and its completion in the private part.

Where Is This Used?

In the process of analyzing, maintaining, or debugging a program, you often need to know where and how various program elements are used. With `Show Usage`, you can trace the usage of program elements throughout the program's units. More specifically, you can use `Show Usage` to:

- List all units in which a given program element is used
- Underline all using occurrences of that element within any of the listed units

To view the using occurrences of a program element such as a subprogram, a variable, or a type:

1. In an Ada unit, put the cursor on any occurrence of this element.
Alternatively, you can select any occurrence of the element; if you do, leave the cursor in the selection.
2. Press `Show Usage`.
One of two actions results:
 - If usages are found only in one Ada unit, these usages are underlined.
 - If usages are found in multiple units, a cross-reference listing of these units (called an *xref*) is displayed in a separate window. Use `Definition` to view these units, in which usages appear underlined.
3. Use `Next Item` and `Previous Item` to move among underlined usages. You can remove the underlines by pressing `Underlines Off`.

`Show Usage` reports occurrences only in units that are installed or coded.

Note that the keyboard overlay lists other related keys in the column labeled "Show." `Show Usage (Unit)` shows using occurrences only in the current unit, whereas `Show Usage (Indirect)` shows using occurrences as well as indirect references in any unit. These keys are bound to variants of the `Ada.Show_Usage` command. For more information, see the Editing Specific Types (EST) book of the *Rational Environment Reference Manual*.

Example 1: Showing Variable References

Assume that you are viewing the declaration of the variable `List_Of_Numbers` in the body of `Display_Complex_Sums`. You want to know where this variable is used, so you select it, as shown in Figure 14-8:

```
with Complex, Complex_Uilities, Text_Io;
procedure Display_Complex_Sums is

    package Complex_List renames Complex_Uilities.Complex_List;

    List_Of_Numbers : Complex_List.List;
    Sum : Complex.Number;
    Input_Line : String (1 .. 100);
    Line_Length : Natural := 0;
    Input_File : Text_Io.File_Type;

    Header : constant String :=
= -- DISPLAY_COMPLEX_SUMS'BODY V.21 -- ada -- installed
```

Figure 14-8. Selecting the Variable `List_Of_Numbers`

Because `List_Of_Numbers` is a local variable within `Display_Complex_Sums'Body`, pressing `Show Usage` underlines all references within that unit, as shown in Figure 14-9:

```
while not Complex_List.Is_Null (List_Of_Numbers) loop
    Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
    Text_Io.Put ("I ");
    Text_Io.Put (Complex_Uilities.Image
                (Complex_List.Head (List_Of_Numbers)));
    Text_Io.Put (" ");
    Text_Io.Put (Complex_Uilities.Image (Sum));
    Text_Io.Put_Line (" I");
    List_Of_Numbers := Complex_List.Tail (List_Of_Numbers);
end loop;
= -- DISPLAY_COMPLEX_SUMS'BODY V.21 -- ada -- installed
```

Figure 14-9. Showing the References to `List_Of_Numbers`

If you scroll back to the declaration of `List_Of_Numbers`, you will notice that this occurrence of the identifier is not underlined. This is because `Show Usage` reports only the using occurrences of an object, not its defining occurrence.

If `List_Of_Numbers` had occurrences in other units, a cross-reference listing of these units would be displayed (see the next section).

Example 2: Showing Usages in Multiple Units

Assume that you are viewing a using occurrence of the function `Complex.Make` in the body of `Display_Complex_Sums`. You want to know where else this function is used, so you select this occurrence, as shown in Figure 14-10:

```

Text_Io.Put_Line ("+" & Header & "+");
Text_Io.Put_Line
  ("|      Statistics on a set of Complex Numbers      |");
Text_Io.Put_Line ("+" & Header & "+");
Text_Io.Put_Line
  ("|      Value      Running Total      |");
Text_Io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_Io.Put ("| ");
= . . . DISPLAY_COMPLEX_SUMS BODY v1201 (ada) Coded

```

Figure 14-10. Selecting a Using Occurrence of `Complex.Make`

Because `Complex.Make` is used in multiple units, pressing `Show Usage` displays a cross-reference listing of these units, as shown in Figure 14-11:

```

Text_Io.Put_Line ("+" & Header & "+");
Text_Io.Put_Line
  ("|      Statistics on a set of Complex Numbers      |");
Text_Io.Put_Line ("+" & Header & "+");
Text_Io.Put_Line
  ("|      Value      Running Total      |");
Text_Io.Put_Line ("|" & Header & "|");
Sum := Complex.Make (0.0, 0.0);

while not Complex_List.Is_Null (List_Of_Numbers) loop
  Sum := Complex.Plus (Sum, Complex_List.Head (List_Of_Numbers));
  Text_Io.Put ("| ");
= . . . DISPLAY_COMPLEX_SUMS BODY v1201 (ada) Coded

```

```

!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX
!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY
!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY
!USERS.ANDERSON.COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS'BODY
= . . . COMPLEX_NUMBERS.COMPLEX_MAKE (xref) FULL_NAMES

```

Figure 14-11. The Cross-Reference Listing for `Complex.Make`

To see the usages of `Complex.Make` in the unit `Complex_Uutilities'Body`, you put the cursor on the entry for that unit in the cross-reference listing and press `Definition`, as shown in Figure 14-12:

```

!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX
!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY
!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY
!USERS.ANDERSON.COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS'BODY

= COMPLEX_NUMBERS.COMPLEX_MAKE (xref) FULL_NAMES

      Imaginary_Part, Scan_Location);
      Scan_Location := Scan_Location + 1;
      Strip_Trailing_Angle (X, Scan_Location);
      return Complex.Make (Real_Part, Imaginary_Part);
exception
  when others =>
    raise Illegal_Complex_Number;
end Value;
end Complex_Uutilities;

= COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY (v1.0) (ada) ~ Coded

```

Figure 14-12. Displaying `Complex_Uutilities'Body` from the Listing

It is important to note that not all units in a cross-reference listing necessarily contain a using occurrence of the specified element. Rather, cross-reference listings are determined by current and former dependencies among whole units; therefore, a given entry in a listing is actually a candidate unit that *potentially* contains a using occurrence.

If you attempt to view a candidate unit that does not contain a using occurrence, its entry is removed from the cross-reference listing. In this example, assume that you use `Definition` to try to view `Complex'Spec` from the cross-reference listing. Because `Complex'Spec` contains only a defining occurrence for `Complex.Make`, pressing `Definition` eliminates the entry for `Complex'Spec` from the listing, as shown in Figure 14-13:

```
Eliminating line - COMPLEX has no references
# Rational (Delta) ANDERSON $ _1
-----
!USERS: ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY
!USERS: ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY
!USERS: ANDERSON.COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS'BODY

# COMPLEX_NUMBERS.COMPLEX_MAKE {xref} FULL_NAMES
-----
        Imaginary_Part, Scan_Location);
    Scan_Location := Scan_Location + 1;
    Strip_Trailing_Angle (X, Scan_Location);
    return Complex.Make (Real_Part, Imaginary_Part);
exception
    when others =>
        raise Illegal_Complex_Number;
end Value;
end Complex_Utilityes;

# COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY v1.51 radst ~ Coded
```

Figure 14-13. Eliminating Complex'Spec from the Listing

You can eliminate all “false” candidates from a cross-reference listing as follows:

1. Put the cursor in the window containing the cross-reference listing.
2. Press `Semanticize`.

Note that this operation can be quite time-consuming for long listings.

Chapter 15. Modifying Installed or Coded Programs

Programs typically require some amount of modification after they have been promoted to the installed or coded state. For example, you may need to change installed or coded units because:

- Testing and debugging have revealed problems that must be corrected.
- New requirements entail corresponding enhancements to program behavior.
- Comments must be added for better program documentation.

One way of changing installed or coded units is to demote them to the source state and make the corrections using basic editing operations. However, if you demote a unit to the source state, you must demote any units that depend on it, as well. Therefore, using this method to change a single unit can potentially entail recompiling a whole program after the change is made.

As an alternative, the Environment provides operations for making certain kinds of changes without having to demote and then repromote units. Using these operations, you can add, change, or delete specific elements in installed or coded units. These operations are called *incremental operations* because they allow individual changes to be compiled incrementally instead of requiring complete recompilation.

This chapter describes:

- The kinds of program elements that can be changed with incremental operations
- How to add, modify, and delete program elements from unit bodies and unit specifications
- How to make changes when incremental operations cannot be used

Elements That Can Be Changed Incrementally

On other computer systems you may have used, the compilation unit is the smallest program element that can be changed and recompiled. On such systems, adding a statement to a unit body requires recompiling the entire body. Similarly, adding a declaration to a unit specification requires recompiling the specification and any other units that depend on it.

In contrast, the Environment allows you to change and recompile program elements at a finer level of granularity. More specifically, you can incrementally add, modify, and delete the following kinds of program elements:

- Stand-alone comments (comments on lines by themselves)
- Statements
- Declarations that have no dependencies
- Context clauses

Thus, on the Environment, the statement or declaration is the smallest program element you can change and recompile. (To change an expression or a field within a record, you must incrementally recompile the whole statement or declaration that contains it.)

If Dependencies Exist

To preserve the validity of your program, the Environment allows you to incrementally change or delete only those program elements that are not referenced by other program elements. Therefore, you can perform incremental operations on any stand-alone comment or statement, because nothing can reference a comment or a statement.

In contrast, declarations can be, and typically are, referenced by other program elements. (Such elements are said to *depend on* the declarations they reference.) For example, a dependency to a declaration is introduced whenever a statement or declaration calls a subprogram or references a variable. Because the Environment manages dependencies on individual declarations (as well as on whole units), it prevents you from incrementally changing or deleting any declaration on which a statement or another declaration already depends.

Whenever you are thus prevented from making a change incrementally, you must demote the relevant units to the source state to make that change (see "Making Changes That Require Demotion," below.)

Units and States

You can use incremental operations to change elements in any installed unit. However, making an incremental change to a coded unit depends on the kind of element you want to change and on the kind of unit that contains the element. Table 15-1 summarizes the permitted states for incremental changes:

Table 15-1. Unit States in Which Incremental Operations Can Be Used

	<i>Unit Body</i>	<i>Unit Specification</i>
<i>Comments</i>	Installed or coded	Installed or coded
<i>Statements and Declarations</i>	Installed only	Installed or coded*

* Automatically demotes the corresponding unit body to installed.

Note that changing statements or declarations in a coded unit specification automatically demotes the corresponding unit body to the installed state. The reason for putting unit bodies in the installed state follows from the fact that incremental changes update a unit's underlying representation, but they do not update the object code associated with that structure. Therefore, old object code must be discarded and new object code must be generated both for changed unit bodies and for unit bodies with changed specifications. (Because comments have no object code, they do not require that units be demoted and then recoded.)

Thus, you can generally leave a unit's state unchanged when you want to use incremental operations, with the following exceptions (at most, you need to recode a body, even if other units are coded against its specification):

- You must demote a coded unit body to installed before you can incrementally change statements or declarations in it.
- You must repromote a unit body to coded after incrementally changing its coded specification.

Using Incremental Operations

The examples in the following sections show how to:

- Modify one or more selected elements using
- Delete one or more selected elements using -
- Add one or more new elements at the cursor position using -

Any of these incremental operations can be applied to comments, statements, and declarations in both unit bodies and specifications, subject to the restrictions noted in the previous sections.

The following examples refer to portions of the coded Ada program `Display_Complex_Sums`. You may find it helpful to refer to Figure 10-3 in Chapter 10 to see how the units in this program are related.

Incrementally Modifying an Element

Assume that you need to correct an error in the coded body of package `Complex`. Specifically, the return statement in the `Minus` function contains plus (+) operators instead of minus (-) operators. Rather than demoting the entire unit to the source state, you can incrementally demote the specific statement to source, edit it, and then promote the statement back into the unit, as shown in the following steps:

1. Adjust the state of `Complex'Body`, if necessary. Because you are incrementally modifying a statement, you must demote the body to installed.

With the cursor in the window containing `Complex'Body`, press `Install Unit` to change the body from coded to installed. Make sure the cursor is not in a selection; otherwise, the `Install Unit` operation will fail.

2. Select the element you want to change—namely, the return statement in the `Minus` function, as shown in Figure 15-1. (Although the desired change affects only part of the statement, you must select the entire statement.)

```

COMPLEX'Body changed to INSTALLED
-----
-- Rationale: (Editor) ANDERSON'S_1
-----
package body Complex is
  function Make (X, Y : Float) return Number is
  begin
    return (X, Y);
  end Make;
  function Real_Part (X : Number) return Float is
  begin
    return X.Real;
  end Real_Part;
  function Imaginary_Part (X : Number) return Float is
  begin
    return X.Imag;
  end Imaginary_Part;
  function Plus (X, Y : Number) return Number is
  begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
  end Plus;
  --
  -- The following function contains a bug
  --
  function Minus (X, Y : Number) return Number is
  begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
  end Minus;
end Complex;
-----
= COMPLEX_NUMBERS_COMPLEX_BODY.vi.63 (ada) (installed)

```

Figure 15-1. Selecting the Statement to Be Modified

3. With the cursor in the selection, press `⌘E` to extract the selected element. As shown in Figure 15-2:
 - A message in the Message window confirms that a statement has been incrementally demoted.
 - A window is opened containing the statement in the source state. This window is a *minor window* because it shares the frame with the major window containing `Complex'Body`. The cursor is put in the minor window.
 - A `[statement]` prompt replaces the extracted statement in `Complex'Body`.

```

Incrementally demoted Statement
= Rational |Delta| ANDERSON $_1
-----
return (X.Real + Y.Real, X.Imag + Y.Imag);

-({STATEMENTS} |ada: | Source
begin
  return (X.Real + Y.Real, X.Imag + Y.Imag);
end Plus;
--
-- The following function contains a bug.
--
function Minus (X, Y : Number) return Number is
begin
  [statement]
end Minus;
end Complex;

= COMPLEX_NUMBERS |COMPLEX BODY |v164| |ada: | Installed

```

Figure 15-2. After Pressing `⌘E`

4. Make the desired changes to the statement (in this case, changing `+` to `-`). You can use basic editing operations and Ada-specific editing operations. Use `⌘F` and `⌘S` to check for errors.

Note that you are not limited to modifying the existing demoted element. In this window, you can enter additional statements, comments, or even levels of structure (for example, by surrounding the demoted statement with an `if` statement).

5. With the cursor in the minor window, press **Promote** to promote the modified statement and return it to Complex'Body, as shown in Figure 15-3. Note that:
- The minor window disappears; its contents replace the [statement] prompt in Complex'Body.
 - A message in the Message window confirms that the statement is now installed.

As usual, **Promote** will fail if there are syntactic or semantic errors.

```
Statement list changed to INSTALLED
= Rational (Delta) ANDERSON S_1
-----
function Plus (X, Y : Number) return Number is
begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
end Plus;
--
-- The following function contains a bug.
--
function Minus (X, Y : Number) return Number is
begin
    return (X.Real - Y.Real, X.Imag - Y.Imag);
end Minus;
end Complex;
-----
COMPLEX_NUMBERS.COMPLEX'BODY.V1651 (ada) Installed
```

Figure 15-3. After Pressing **Promote**

6. Press **Promote** again to recode Complex'Body. The program can now be executed.

Selecting One or More Elements

You can use any selection operation to designate one or more elements for incremental modification (or deletion, described below). The following guidelines may be helpful:

- Use object selection operations (such as **Object** - **[-]**) to select a single element at any level of program structure. An entire list of statements or declarations at the same level counts as a single element. For example, in Complex'Body, you can use **Object** - **[-]** to select the entire Minus function or the entire list of functions, from Make to Minus.
- Use **Region** - **[|]** and **Region** - **[|]** to select multiple elements at the same level, such as a partial list of statements or declarations. For example, in Complex'Body, you can use these operations to select just the two functions Plus and Minus.

Using the Window Provided by an Incremental Operation

A minor window is opened when an element is incrementally modified (or added; see below). The window banner identifies the element you are modifying or adding—in Figure 15-2, the minor window is identified as a [STATEMENTS] window.

This window is logically connected to the prompt that marks its place in the parent unit. For example, if you delete the prompt, the window and its contents are discarded. (See “Incrementally Deleting an Element,” below.)

If you cannot repromote the contents of the window because of errors, you can use **Enter** to save your work until the errors are corrected. (You can use **Format** and **Semanticize** for syntactic and semantic assistance.) The demoted portion of the unit appears in the library with a temporary name (such as `_Ada_4_`), which is listed under the parent unit’s name.

To enlarge a minor window, you can make it into a major window by pressing **Window - Promote**. Note that you can use **Definition** and **Enclosing** to traverse between the prompt in the parent unit and the element you are incrementally editing.

Incrementally Deleting an Element

Having corrected the error in the Minus function, you now delete the comment lines above it, as shown in the following steps. You can use incremental deletion here because these are stand-alone comment lines rather than right-trailing comments.

1. Check the state of Complex’Body. Because you are incrementally deleting a comment, you can leave Complex’Body in the coded state.
2. Select the element you want to delete—namely, the comment above the Minus function, as shown in Figure 15-4:

```

COMPLEX'Body changed to CODED
= Rational {Delta} ANDERSON S_1
-----
function Plus (X, Y : Number) return Number is
begin
    return (X.Real + Y.Real, X.Imag + Y.Imag);
end Plus;
-----
-- The following function contains a bug
-----
function Minus (X, Y : Number) return Number is
begin
    return (X.Real - Y.Real, X.Imag - Y.Imag);
end Minus;
end Complex;

= COMPLEX_NUMBERS COMPLEX'BODY v:65: (ada) Coded

```

Figure 15-4. Selecting the Comment to Be Deleted

Incrementally Adding an Element

Assume that you want to add a new function called `Times` to package `Complex`. To do this, you need to add a new declaration to `Complex'Spec` and then build a corresponding function body within `Complex'Body`, as shown in the following sections.

Adding a New Declaration

The following steps show how to add a declaration for the `Times` function to `Complex'Spec`. You can leave `Complex'Spec` in the coded state because it is a specification. (Note that modifying `Complex'Spec` automatically changes `Complex'Body` to the installed state.)

1. In `Complex'Spec`, position the cursor after the last function declaration, as shown in Figure 15-6:

```
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number; ■
private
  type Number is
    record
      Real, Imag : Float;
    end record;
end Complex;
```

COMPLEX_NUMBERS COMPLEX.V13 | ada | Coded

Figure 15-6. Designating the Location of the New Declaration

2. Press **Object** - **I**. As shown in Figure 15-7:

- A minor window is opened with a [declaration] prompt in the source state. The banner under this window identifies it as a [DECLARATIONS] window.
- A matching [declaration] prompt appears in Complex'Spec where the new declaration will be inserted. This [declaration] prompt is automatically selected; that is, it is both in reverse video and in the selection font.

```

[declaration]
-----
# [DECLARATIONS] (ada) Source
function Make (X, Y : Float) return Number;
function Real_Part (X : Number) return Float;
function Imaginary_Part (X : Number) return Float;
function Plus (X, Y : Number) return Number;
function Minus (X, Y : Number) return Number;
[declaration]
private
type Number is
-----
# COMPLEX_NUMBERS_COMPLEX_V14 (ada) Coded

```

Figure 15-7. After Pressing **Object** - **I**

3. With the cursor on the [declaration] prompt in the minor window, enter the desired declaration, as shown in Figure 15-8. Use **Format** and **Semanticize** for completion and error checking. You can add arbitrarily many declarations and comment lines.

```

--
-- This function is for demonstration purposes only:
function Times (X, Y : Number) return Number;
--
-----
# [DECLARATIONS] (ada) Source
function Make (X, Y : Float) return Number;
function Real_Part (X : Number) return Float;
function Imaginary_Part (X : Number) return Float;
function Plus (X, Y : Number) return Number;
function Minus (X, Y : Number) return Number;
[declaration]
private
type Number is
-----
# COMPLEX_NUMBERS_COMPLEX_V14 (ada) Coded

```

Figure 15-8. Entering a New Declaration and Comment Lines

4. With the cursor in the minor window, press **Promote** to promote the new declaration into Complex'Spec, as shown in Figure 15-9. As a result:
 - The minor window disappears and its contents replace the [declaration] prompt in Complex'Spec.
 - A message in the Message window confirms that the new declaration is now coded.

```

Declaration list changed to CODED
= Rational |Delta| ANDERSON S_1
-----
function Make (X, Y : Float) return Number;
function Real_Part (X : Number) return Float;
function Imaginary_Part (X : Number) return Float;
function Plus (X, Y : Number) return Number;
function Minus (X, Y : Number) return Number;
--
-- This function is for demonstration purposes only:
function Times (X, Y : Number) return Number;
--
private
  type Number is
-----
= << COMPLEX_NUMBERS COMPLEX 'v,15 | r ada | Coded
-----

```

Figure 15-9. After Pressing **Promote**

Adding the Corresponding Body

The following steps show how to incrementally build a body for the Times function from its declaration in Complex'Spec. The new body is inserted in the correct order in Complex'Body. As a result of adding the new declaration to Complex'Spec, Complex'Body has already been changed to the installed state.

1. Select the declaration for Times in Complex'Spec, as shown in Figure 15-10.

```

-----
function Make (X, Y : Float) return Number;
function Real_Part (X : Number) return Float;
function Imaginary_Part (X : Number) return Float;
function Plus (X, Y : Number) return Number;
function Minus (X, Y : Number) return Number;
--
-- This function is for demonstration purposes only:
function Times (X, Y : Number) return Number;
-----
private
  type Number is
-----
= COMPLEX_NUMBERS COMPLEX 'v,15 | r ada | Coded
-----

```

Figure 15-10. Selecting the Declaration for the Times Function

2. Press **Create Body**. As shown in Figure 15-11:

- **Complex'Body** is automatically displayed.
- A skeletal function body for **Times** is displayed in a minor window above **Complex'Body**.
- A **[declaration]** prompt is inserted at the end of **Complex'Body**, marking the location of the new function body.

```

function Make (X, Y : Float) return Number;
function Real_Part (X : Number) return Float;
function Imaginary_Part (X : Number) return Float;
function Plus (X, Y : Number) return Number;
function Minus (X, Y : Number) return Number;
--
-- This function is for demonstration purposes only
function Times (X, Y : Number) return Number;
--

```

```

COMPLEX_NUMBERS COMPLEX.V151 (ada) Coded

```

```

function Times (X, Y : Number) return Number is
begin
  [statement]
end Times;

```

```

[DECLARATIONS] (ada) Source
function Minus (X, Y : Number) return Number is
begin
  return (X.Real - Y.Real, X.Imag - Y.Imag);
end Minus;
[declaration]
end Complex,

```

```

COMPLEX_NUMBERS COMPLEX'BODY.V167 (ada) Installed

```

Figure 15-11. After Pressing **Create Body**

3. With the cursor on the **[statement]** prompt in the minor window, complete the function body for **Times**. Use **Format** and **Semanticise** for syntactic completion and semantic checking.

4. With the cursor in the minor window, press **Promote** to promote the new body into Complex'Body, as shown in Figure 15-12. As before, the minor window disappears and a message is displayed in the Message window.

```

No semantic errors
Declaration list changed to INSTALLED
= Rational ↵Delta, ANDERSON S_1
-----
function Make (X, Y : Float) return Number;
function Real_Part (X : Number) return Float;
function Imaginary_Part (X : Number) return Float;
function Plus (X, Y : Number) return Number;
function Minus (X, Y : Number) return Number;
--
-- This function is for demonstration purposes only
function Times (X, Y : Number) return Number;
--
-----
COMPLEX_NUMBERS.COMPLEX V.15, (ada) Coded
-----
function Minus (X, Y : Number) return Number is
begin
    return (X.Real - Y.Real, X.Imag - Y.Imag);
end Minus;
function Times (X, Y : Number) return Number is
begin
    return (X.Real * Y.Real, X.Imag * Y.Imag);
end Times;
end Complex;
-----
COMPLEX_NUMBERS.COMPLEX BODY V.68, (ada) Installed
-----

```

Figure 15-12. Promoting the Body for the Times Function into Complex'Body

5. Press **Promote** again to recode Complex'Body. The program can now be executed.

Determining the Kind of Element That Is Added

The location of the cursor in a unit determines what kind of element you are prompted for when you press **Object** - **I**. Putting the cursor in a declaration list produces a [declaration] prompt in a window labeled [DECLARATIONS]. Likewise, putting the cursor in a statement list produces a [statement] prompt in a window labeled [STATEMENTS]. If you try to add a statement where a declaration is expected, the statement is syntactically completed as a declaration, and vice versa. (If this happens, delete the prompt and start over.)

Certain other constructs, such as exception handlers, can be added incrementally using special cursor placement. For example, you can add an exception handler to the body of the Times function by putting the cursor at the beginning of the end statement for that body, as shown in Figure 15-13:

```
function Minus (X, Y : Number) return Number is
begin
  return (X.Real - Y.Real, X.Imag - Y.Imag);
end Minus;
function Times (X, Y : Number) return Number is
begin
  return (X.Real * Y.Real, X.Imag * Y.Imag);
  ─ end Times;
end Complex;
```

COMPLEX_NUMBERS.COMPLEX BODY v1.77 | ada | Installed

Figure 15-13. Positioning the Cursor to Add an Exception Handler to Times

When you press `Object` - `I`, the reserved word `exception` is added, followed by a prompt for an alternative. The corresponding minor window also contains an `[alternative]` prompt, as shown in Figure 15-14:

```
[alternative]
```

```
ALTERNATIVES | ada | Source
end Minus;
function Times (X, Y : Number) return Number is
begin
  return (X.Real * Y.Real, X.Imag * Y.Imag);
exception
  [alternative]
end Times;
```

COMPLEX_NUMBERS.COMPLEX BODY v1.78 | ada | Installed

Figure 15-14. After Pressing `Object` - `I`

Note that placing the cursor at the end of the previous statement (the return statement in Times) would have resulted in a `[statement]` prompt instead of an exception handler.

Some Common Problems

The following sections discuss common problems that can arise when you use incremental operations.

Removing an Unwanted Prompt

After using `[Object] - [I]`, you may decide that the resulting prompt is in the wrong place or prompts for the wrong kind of program element. For example, if the cursor was positioned incorrectly, an exception handler may appear when you wanted a `[statement]` prompt. To remove an unwanted prompt:

1. Select the prompt in the major window, if it is not already selected. (In some cases, the prompt is automatically selected when it is first inserted.)
2. Press `[Object] - [D]` to delete the prompt and remove the corresponding minor window and its contents.

Deleting the `[alternative]` prompt in an exception handler automatically deletes the reserved word `exception`. In fact, when a construct like an exception handler consists of a reserved word followed by a list of items, the only way to delete the reserved word is to delete the list below it. (You cannot select and delete the reserved word directly.)

Forgetting to Demote a Body

Unit bodies must be in the installed state before statements or declarations can be added, modified, or deleted. If you forgot to demote a body to installed, a message like the following is displayed in the Message window when you try to complete the incremental operation:

```
1: ERROR Incremental operations in coded units are only allowed for
library unit package specs
```

If this message is displayed:

1. Turn off any selections in the unit body.
2. With the cursor in the window containing the unit body, press `[Install Unit]`.
3. Repeat or continue the desired incremental operation.

Selecting a Construct That Cannot Be Edited

Not every selectable construct can be edited or deleted. For example, assume that you want to change the names of the parameters in the Plus function. If you select just the parameter list and press **Edit**, a message in the Message window reports that this construct cannot be edited, as shown in Figure 15-15:

```
Demote failed - "X, Y : Number" is not an editable construct.
```

```

Rational (Delta) ANDERSON S_1
-----
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
  --
COMPLEX_NUMBERS COMPLEX 'V(15) (ada) Coded

```

Figure 15-15. Attempting to Edit a Parameter List

Attempting to Change a Declaration That Has Dependents

You cannot incrementally edit or delete a declaration on which other program elements depend. For example, assume that you select the entire declaration for the Plus function in order to change its parameter names. However, because Plus is referenced by statements or declarations in other units, pressing **Edit** causes an *obsolescence menu* to be displayed, as shown in Figure 15-16:

```
Demote failed - would obsolesce other units
```

```

Rational (Delta) ANDERSON S_1
-----
package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (X, Y : Number) return Number;
  function Minus (X, Y : Number) return Number;
  --
COMPLEX_NUMBERS COMPLEX 'V(15) (ada) Coded
-----
Units that are obsolesced by PLUS'Spec
!USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY'V(68)
!USERS.ANDERSON.COMPLEX_NUMBERS.DISPLAY_COMPLEX_SUMS'BODY'V(21)

```

```
(menu)
```

Figure 15-16. Attempting to Edit Plus, Which Has Dependents

This menu lists the names of all units that contain references to Plus. The named units therefore need to be recompiled when Plus is changed. You can traverse to a listed unit by pressing **Definition**. You can get more specific information (each reference underlined) by selecting Plus and pressing **Show Usage**.

Making Changes That Require Demotion

Some changes cannot be made incrementally because they require the recompilation of other units or other portions of the same unit. Such recompilation is required to modify or delete a declaration on which other program elements depend. If you try to incrementally change or delete a declaration that has dependents, an obsolescence menu is displayed, as described in the previous section.

When an incremental operation produces an obsolescence menu, you can use the automatic compilation facilities to demote and repromote the affected units. For example, the following steps show how to change the names of the parameters in the Plus function. Changes need to be made in both Complex'Spec and Complex'Body.

1. Select the library entry for Complex'Spec or, if the cursor is in the window containing Complex'Spec, select the entire unit by repeatedly pressing **Object** - **[-]**.
2. Press **Source (This World)**. As shown in Figure 15-17, a log is displayed showing the dependent units that are now demoted to the source state:

```

-----
ANDERSON.COMPLEX_NUMBERS.COMPLEX'V(16) % COMPILATION.DEMOTE STARTED 9:41:03 PM
-----
87/08/29 21:41:04 ::: [Compilation.Demote ("<SELECTION>", SOURCE, "<WORLDS>",
87/08/29 21:41:05 ... FALSE, PERSEVERE);].
87/08/29 21:41:05 --- Attempting to demote !USERS.ANDERSON.COMPLEX_NUMBERS.
87/08/29 21:41:05 ... COMPLEX.
87/08/29 21:41:05 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX'BODY demoted to
87/08/29 21:41:05 ... SOURCE.
87/08/29 21:41:06 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES.
87/08/29 21:41:06 ... IMAGE'BODY demoted to SOURCE.
87/08/29 21:41:06 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES'BODY
87/08/29 21:41:06 ... demoted to SOURCE.
87/08/29 21:41:06 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.
87/08/29 21:41:06 ... DISPLAY_COMPLEX_SUMS'BODY demoted to SOURCE.
87/08/29 21:41:06 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX_UTILITIES
87/08/29 21:41:06 ... demoted to SOURCE.
87/08/29 21:41:07 +++ !USERS.ANDERSON.COMPLEX_NUMBERS.COMPLEX demoted to
87/08/29 21:41:07 ... SOURCE.
87/08/29 21:41:07 ::: [End of Compilation.Demote Command].

```

```

-----
COMPLEX'V(16) % COMPILATION.DEMOTE (text) ~
-----

```

Figure 15-17. Demoting Complex'Spec to the Source State

3. After opening the appropriate units for editing, make the desired changes. In this case, change the parameter names *X* and *Y* to *Left* and *Right* in both *Complex'Spec* and *Complex'Body*, as shown in Figure 15-18:

```

package Complex is
  type Number is private;
  function Make (X, Y : Float) return Number;
  function Real_Part (X : Number) return Float;
  function Imaginary_Part (X : Number) return Float;
  function Plus (Left, Right : Number) return Number;
  function Minus (X, Y : Number) return Number;
  --
# COMPLEX_NUMBERS COMPLEX.V:24, 1 ada, ~ Source
begin
  return X.Imag;
end Imaginary_Part;
function Plus (Left, Right : Number) return Number is
begin
  return (Left.Real + Right.Real, Left.Imag + Right.Imag);
end Plus;
function Minus (X, Y : Number) return Number is
begin
  return (X.Real - Y.Real, X.Imag - Y.Imag);
end Minus;
# COMPLEX_NUMBERS COMPLEX.BODY.V:105, 1 ada, ~ Source

```

Figure 15-18. Changing the Names of the Parameters in the Plus Function

4. Repromote the program to the coded state. In this case, select the library entry for the program's main procedure, *Display_Complex_Sums*, and press .

Index

! (root world)	I-28
! banner symbol	I-48
!Commands	I-29
!Io	I-29
!Lrm	I-29
!Tools	I-29
!Users	I-29
# banner symbol	I-48
* banner symbol	I-48
= banner symbol	I-48
@ banner symbol	I-47
~ banner symbol	I-47, I-51
-- comment character	III-47

A

abbreviations	
command names	I-76
package names	I-81
access control	I-28
Activate key (Debugger)	III-68
activity files	I-25
Ada block statement	I-71
Ada compilation units	I-26
changing unit state	III-25
closing source units for editing	III-20

Ada compilation units, <i>continued</i>	
creating	III-13
determining unit names and subclasses	III-14
package specifications and bodies	III-15
subprograms	III-15
defined	III-2
editing	II-9, III-27
checking for semantic errors	III-36
creating bodies	III-45
creating private parts	III-43
entering comments	III-47
Format key	III-27
inserting page breaks	III-48
selecting Ada constructs	III-39
Environment compilation system	III-5
opening existing units for editing	III-18, III-19
write locks	III-18
overview of development	III-1
sample library	III-6
sample procedure	III-7
promoting to coded state	III-49
coding individual units	III-49
coding units with dependencies	III-50
promoting to installed state	III-20
installing units with dependencies	III-22
reading the compilation log	III-24
saving work in progress	III-17
discarding unsaved changes	III-17
testing	III-53
unit states	III-3
archived	III-3
coded	III-4
installed	III-3
source	III-3
Ada programs	
browsing	III-79
definition	III-80
options	III-83
selection versus cursor position	III-83
showing usage	III-86
debugging	III-57
automatic source display	III-60
catching exceptions	III-73
controlling program execution	III-60
Debugger window	III-59
displaying variable values	III-70
modifying variable values	III-71
redisplaying current location	III-71
reexecuting a program	III-71

Ada programs, <i>continued</i>	
debugging, <i>continued</i>	
setting breakpoints	III-67
starting the Debugger	III-58
stepping through a program	III-61
examining the stack of subprogram calls	III-74
executing	III-51
common errors	III-52
job control operations	III-52
using a Command window	III-51
using selection	III-52
modifying installed or coded	III-91
adding an element	III-99
common problems	III-105
deleting an element	III-97
incrementally changeable elements	III-92
making changes that require demotion	III-107
modifying an element	III-94
using incremental operations	III-93
testing	III-53
saving interactive test programs	III-55
Ada.Show_Usage command	III-86
Ada-specific editing operations	III-27
checking for semantic errors	III-36
creating bodies	III-45
creating private parts	III-43
entering comments	III-47
Format key	III-27
entering a function	III-28
hints for using	III-32
inserting page breaks	III-48
selecting Ada constructs	III-39
syntactic and semantic error reporting	III-37
testing programs	III-54
Ada syntax rules	I-69
Ada usage, in Command windows	I-69
entering parameters	I-69
adding text, <i>see</i> editing text	
archived state	III-3
arrow keys	I-5
attribute	I-30
auxiliary keys	I-6

B

bang (!)	I-28
banner, window	I-22
fields	I-47
modification symbols	I-48
Begin Of key	I-5, I-46
binary files	I-25
block statement	I-71
bodies, creating	III-15, III-45
Break key (Debugger)	III-67
Break Default key (Debugger)	III-68
breakpoints	
characteristics	III-68
activated	III-68
permanent	III-68
temporary	III-68
executing to a breakpoint	III-69
setting	III-67
browsing	III-79
definition	III-80
options	III-83
selection versus cursor position	III-83
showing usage	III-86
<i>see also</i> moving; traversing	

C

call stack	III-74
case	
changing character case	II-20
in key notation	I-11
Catch key (Debugger)	III-74
changing	
number of frames	I-60
password	I-17
placement and size of windows	I-56
terminal type	I-16
characters	II-10
classes of objects	I-25
Ada compilation units	I-26
files	I-25
libraries	I-27

<code>Code (All Worlds)</code> key	III-26
<code>Code (This World)</code> key	III-26, III-49
<code>Code Unit</code> key	III-26, III-49
coded programs, modifying, <i>see</i> Ada programs	
coded state	III-4, III-49
coding	
individual units	III-49
units with dependencies	III-50
comma key, numeric keypad	I-4
Command window	I-17, I-23, I-63
Ada block statement	I-71
Ada usage	I-69
attached windows	I-68
clearing	I-76
editing	II-9
executing programs	III-51
history	I-79
keys	I-79
recalling previous commands	I-79
reusing	I-77
testing through	III-53
using	I-64
visibility	I-81
commands	I-80
abbreviating	I-76
Ada and Environment names	I-80
canceling execution	I-68
completing ambiguous name fragments	I-74
correcting typing errors	I-68
executing	I-63
subsequent commands	I-77
filling in parameter prompts	I-73
getting prompting assistance	I-72
formatting	I-72
semantic completion	I-72
modifying and reexecuting	I-78
reexecuting the same command	I-77
unrecognized	I-67
comments, entering	III-47
commit	I-48, II-4
Common.Definition command	I-64, I-68, I-69, I-70, I-72, I-84, I-85, III-20
Common.Revert command	II-4, III-17

compilation, incremental, <i>see</i> incremental operations	
Complete key	I-6, I-72, I-74
completion, semantic	
ambiguous name fragments	I-74
getting prompting assistance	I-72
menu entries	I-75
composing mode	II-17
Control key	I-5, I-7, I-8, II-11, II-19, II-20
controlling case and text format, <i>see</i> editing text	
conventions, in key notation	I-10
copying text, <i>see</i> editing text	
Create Ada key	III-7, III-14
Create Body key	III-16, III-45
Create command	
Io.Create	I-73
Text.Create	I-25, II-1, II-3
Create Command key	I-17, I-77
Create Private key	III-15, III-43
Create Text key	II-1
creating Ada units, <i>see</i> Ada compilation units	
current context	I-30
cursor keys	I-5

D

Debug.Modify command	III-71
Debugger	III-57
catching exceptions	III-73
controlling program execution	III-60
automatic source display	III-60
stepping through a program	III-61
displaying variable values	III-70
examining the stack of subprogram calls	III-74
displaying parameter values for a frame	III-77
displaying qualified names in the stack	III-76
displaying the call stack	III-74
traversing from the call stack	III-76
modifying variable values	III-71
redisplaying current location	III-71

Debugger, <i>continued</i>	
reexecuting a program	III-71
setting breakpoints	III-67
characteristics	III-68
executing to a breakpoint	III-69
starting	III-58
Debugger window	III-59
window	III-59
<code>Debugger Window</code> key	III-71
debugging Ada programs, <i>see</i> Debugger	
default access classes	I-28
default session	I-14
default window placement	I-51
defining occurrence, displaying	III-80
Definition command	
Common.Definition	I-64, I-68, I-69, I-70, I-72, I-84, I-85, III-20
<code>Definition In Place</code> key	I-52, I-69
<code>Definition</code> key	I-31, I-32, I-36, I-40, I-55, III-79
<code>Delete</code> key	I-4, I-13, I-56
deleting	
text, <i>see</i> editing text	
windows, <i>see</i> windows	
<code>Demote</code> key	I-52, III-25
dependencies	
coding units	III-50
in incremental compilation	III-92
directories	I-27
<i>see also</i> libraries; worlds	
displaying terminal type	I-16

E

<code>Edit</code> key	I-77, II-6, III-93
editing text	II-9
adding text	II-9
closing files for editing	II-7
controlling case and text format	II-20
adjusting text format	II-21
centering lines	II-23
changing character case	II-20
changing fill column	II-23

editing text, <i>continued</i>	
controlling case and text format, <i>continued</i>	
filling existing lines of text	II-22
inserting page breaks	II-23
justifying text	II-22
setting word wrap	II-21
copying	II-15
deleting	II-14
duplicating a line	II-15
moving	II-16
opening an existing file	II-5
patterns in operations	II-11
retrieving deleted text	II-14
saving changes	II-4
searching and replacing	II-17
summary (table)	II-19
selecting text items	II-11
arbitrary stretch of text	II-11
summary (table)	II-13
turning off selection	II-13
using object selection	II-12
within a file's hierarchy	II-12
summary of copy and move (table)	II-16
transposing	II-16
write locks	II-6
<i>see also</i> Ada-specific editing operations	
Editor.Quit command	I-18, I-48
<code>Enclosing In Place</code> key	I-52
<code>Enclosing</code> key	I-31, I-40, I-52
<code>End Of</code> key	I-5, I-46
<code>Enter</code> key	I-4, I-55, II-4
<i>see also</i> commit	
errors	
correcting typing	I-68
in program execution	III-52
exceptions	
catching	III-73
propagating	III-74
Standard.Program_Error	III-21, III-52
executing	
Ada programs	III-51
commands	I-63

executing, <i>continued</i>	
key combinations	I-6
previous commands	I-79
Explain key	I-23, I-67, III-10

F

files, text	I-25
closing for editing	II-7
creating	II-1
editing	II-9
entering text	II-3
opening for editing	II-5
saving changes	II-4
write locks	II-6
fill	
column	II-21
mode	II-21
Format key	I-6, III-8, III-27, III-32
formatting	I-48, II-20
adjusting	II-21
getting prompting assistance	I-72
frames	I-49, III-74
changing number	I-60
joining	I-56
making sizes equal	I-58
fully qualified pathname	I-30
function keys	I-4

H

help	
getting	I-83
help keys	I-83
Help window	I-83, I-85
reading help menus	I-87, I-88
reading help messages	I-84
Help On Help key	I-83
Help On Key key	I-84
Help window	I-83, I-85
reading menus	I-87, I-88
Help Window key	I-85
hold stack	II-14

home	
library	I-22
world	I-27
Home Library key	I-31, I-38, I-52
I	
Image key	I-5, I-24, I-46
Image_Fill_Column session switch	II-23
Image_Fill_Mode session switch	II-21
Image_Insert_Mode session switch	II-10
images	I-45
scrolling	I-45
incremental compilation, <i>see</i> incremental operations	
incremental operations	III-4, III-91
adding an element	III-99
common problems	III-105
deleting an element	III-97
incrementally changeable elements	III-92
dependencies	III-92
units and states	III-93
making changes that require demotion	III-107
modifying an element	III-94
using	III-93
insert mode	II-9
Install (All Worlds) key	III-26
Install (This World) key	III-22, III-26
Install Unit key	III-20, III-26
installed programs, modifying, <i>see</i> Ada programs; incremental operations	
installed state	III-3, III-20
interactive testing, Ada programs	III-53
Io.Create command	I-73
Io.Put command	III-54
item keys	I-5, I-6
Item Off key	I-73, I-78, III-39
item-operation key combinations	I-6, II-11

J

job control III-52

K

key bindings I-1

key combinations I-63, II-11

 and Command windows I-79

 executing I-6

 item-operation key combinations I-6

 modified key combinations I-7

key names I-1, I-6, I-8

 logical I-8

key notation, summary I-10

keyboard, Rational Terminal I-1

 layout I-2

 auxiliary keys I-6

 cursor keys I-5

 function keys I-4

 item keys I-5

 main keyboard I-4

 modifier keys I-5

 numeric keypad I-4

 overlay I-8

 how to read I-8

 organization I-8

 summary of key notation I-10

keycaps I-6, I-8

keys, *see* individual key names; keyboard, Rational Terminal

L

layout, Rational Terminal keyboard I-2

libraries I-22, I-27

 current context I-30

 customized display format I-23

 directories I-27

 pathnames I-30

 structure I-28

 traversing I-31

 worlds I-27

Library_Show_Standard session switch I-23

Library_Std_Show_Unit_State session switch I-23

Line key I-5, I-6, II-11, II-20

lines	II-10
locking windows	I-51
logical key names	I-8
login	I-13
basic process	I-13
changing password	I-17
customized display	I-24
multiple sessions	I-15
nondefault sessions	I-15
terminal type	I-16
what you see	I-21
logout	I-18
with unsaved changes	I-18

M

main keyboard	I-4
major window	I-22
managing windows	I-45
Mark key	I-5
menu	
completion	I-75
obsolescence	III-18, III-106
menu window	I-74
Message window	I-22
Meta key	I-5, I-7, I-8, II-19
minor window	I-45, III-95
minus key, numeric keypad	I-4
modification symbols, in window banner	I-48
modified key combinations	I-7
modifier keys	I-5
Modify command	
Debug.Modify	III-71
Modify key	III-71
modifying	
installed or coded programs, <i>see</i> Ada programs	
text, <i>see</i> editing text	

moving	
between and within windows	I-24, I-50
between prompts	I-73
in the Environment	I-21
text, <i>see</i> editing text	
within the library hierarchy	I-31
<i>see also</i> browsing; traversing	
multiple sessions	
login	I-15

N

name components	I-30, I-80
name fragments, completing	I-74
names of objects	I-30
negative numeric arguments, key notation	I-11
<code>Next Item</code> key	I-73, II-19, III-38
<code>Next Prompt</code> key	I-73
<code>Next Underline</code> key	III-38
nondefault sessions	
login	I-15
notation conventions, key	I-10
number keys, numeric keypad	I-4
numeric arguments, key notation	I-10
numeric keypad	I-4

O

object classes	I-25
Ada compilation units	I-26
files	I-25
libraries	I-27
<code>Object</code> key	I-5, I-59, I-79, II-4, II-6, III-39, III-93
object names	I-30
objects	
abandoning	I-59
releasing	I-59
obsolescence menu	III-18
on-line help	I-83
operation keys	I-6

Other Part key	I-31, I-36, I-52
overlay, Rational Terminal keyboard	I-8
how to read	I-8
organization	I-8
overwrite mode	II-9
P	
package specifications, creating	III-15
page breaks, inserting	
Ada compilation units	III-48
text files	II-23
paragraphs	II-10
parameter placeholders	I-81
parameters	
entering	I-69
filling in prompts	I-73
password	I-13, I-14
changing	I-17
temporary	I-17
pathnames	I-30
period key, numeric keypad	I-4
placement of windows	I-49
changing placement and size	I-56
default	I-51
Previous Item key	I-73, II-19, III-38
Previous Prompt key	I-73
Previous Underline key	III-38
private parts, creating	III-43
Program_Error exception	
Standard.Program_Error	III-21, III-52
programs, Ada, <i>see</i> Ada programs	
Promote key	I-6, I-17, II-4, II-6, III-25, III-49
promoting Ada units, <i>see</i> Ada compilation units	
prompt	I-64
filling in parameter prompts	I-73
getting prompting assistance	I-72
formatting	I-72, III-27
semantic completion in Command windows	I-72
in incremental operations	III-95, III-100, III-104, III-105
moving between prompts	I-73

Prompt For key	I-79
Propagate key (Debugger)	III-74
Put command	
Io.Put	III-54

Q

Quit command	I-55
Editor.Quit	I-18, I-48

R

Rational Terminal keyboard	I-1
layout	I-2
auxiliary keys	I-6
cursor keys	I-5
function keys	I-4
item keys	I-5
main keyboard	I-4
modifier keys	I-5
numeric keypad	I-4
overlay	I-8
how to read	I-8
organization	I-8
summary of key notation	I-10

read access	I-28
------------------------------	------

rearranging windows, *see* windows

Region key	I-5, II-11, II-20, III-39
-----------------------------	---------------------------

Remove Breaks key (Debugger)	III-68
---	--------

removing windows, *see* windows

Repeat key	I-4
-----------------------------	-----

reserved words	III-14
---------------------------------	--------

Return key	I-4, I-13
-----------------------------	-----------

Revert command	
Common.Revert	II-4, III-17

root task	III-59
----------------------------	--------

root world	I-28
-----------------------------	------

S

saving	
Ada editing	III-17
changes, <i>see</i> commit	
interactive test programs	III-55
text editing	II-4

scrolling an image	I-45
searching and replacing text, <i>see</i> editing text	
searchlist	I-81
selecting text, <i>see</i> editing text	
selection	
Ada units	III-41
executing programs	III-52
text items	II-11
arbitrary stretch of text	II-11
summary (table)	II-13
turning off	II-13
using object selection	II-12
within a file's hierarchy	II-12
versus cursor position, in browsing	III-83
semantic completion	
getting prompting assistance	I-72
semantic errors	
checking for	III-36
reporting	III-37
Semanticize key	III-10, III-36
sentences	II-10
session name	I-14, I-15
session switches	I-23
Image_Fill_Column	II-23
Image_Fill_Mode	II-21
Image_Insert_Mode	II-10
Library_Show_Standard	I-23
Library_Std_Show_Unit_State	I-23
Word_Breaks	II-10
sessions	I-14, I-15
Shift key	I-5, I-7, I-8
Show Breaks key (Debugger)	III-68
Show Source key (Debugger)	III-71
Show Usage (Indirect) key	III-86
Show Usage (Units) key	III-86
Show Usage key	III-79
Show_Usage command	
Ada.Show_Usage	III-86
showing usage	III-86

simple name	I-30
Source (All Worlds) key	III-26
Source (This World) key	III-26
source state	III-3
Source Unit key	III-26
special names	I-81
Stack key (Debugger)	III-74
stepping operations, Debugger	III-61
structure of libraries	I-28
subclasses	I-25
switch files	I-25
switches, session, <i>see</i> session switches	
symbols	
key notation	I-10
window banner	I-48
syntactic errors	
reporting	III-37
systems, testing	III-53

T

temporary password	I-17
terminal type	
changing	I-16
checking at login	I-16
displaying	I-16
text files	I-25
closing for editing	II-7
creating	II-1
editing	II-9
entering text	II-3
opening for editing	II-5
saving changes	II-4
write locks	II-6
Text.Create command	I-25, II-1, II-3
Text.Write_File command	III-59
transposing	
text, <i>see</i> editing text	
windows, <i>see</i> windows	

traversing	
between Ada specifications and bodies	I-36
Environment library structure	I-31
from library to object in it	I-32
Rational Environment	I-21
returning to home library	I-38
summary	I-42
to the enclosing library	I-40
<i>see also</i> browsing; moving	

U

<code>Unicode (All Worlds)</code> key	III-26
<code>Unicode (This World)</code> key	III-26
<code>Underlines Off</code> key	III-38
unit states	III-2, III-3
archived	III-3
changing	III-25
coded	III-4
installed	III-3
source	III-3
unsaved changes, logout	I-18
usage, showing	III-86
user account	I-13
username	I-13, I-14
using Command windows	I-64

V

variable values, Debugger	
displaying	III-70
modifying	III-71
versions	
Ada compilation units	III-19
files	II-2, II-4
visibility, in Command windows	I-81

W

Window Directory	I-52
checking before logout	I-55
displaying	I-53
redisplaying replaced windows	I-54
<code>Window</code> key	I-5, I-6, I-24, I-50, I-51, I-52, I-56

Window.Frames command	I-60
windows	I-22
and frames	I-49
and images	I-45
banner in	I-22
fields	I-47
modification symbols	I-48
changing number of frames	I-60
changing size and placement	I-56
Command	I-23
Ada block statement	I-71
Ada usage	I-69
attached windows	I-68
editing	II-9
entering parameters	I-69
executing programs	III-51
testing through	III-53
using	I-64
visibility	I-81
controlling placement	
locking	I-51
traversal commands	I-52
Debugger	III-59
editing	II-9
expanding	I-57
Help	I-83, I-85
joining frames	I-56
major	I-22
making frames equal	I-58
managing	I-45
menu	I-74
Message	I-22
minor	III-95
moving between and within	I-24, I-50
placement on screen	I-49
default	I-51
rearranging	I-59
redisplaying replaced windows	I-54
redisplaying using Window Directory	I-52
removing	I-58
scrolling an image	I-45
shrinking	I-58
transposing	I-59
Word key	I-5, I-6, II-11, II-20
word wrap, setting	II-21
Word_Breaks session switch	II-10
words	II-10

world !	I-28
worlds	I-27
<i>see also</i> libraries	
write access	I-28
write locks	
Ada compilation units	III-18
text files	II-6
Write_File command	
Text.Write_File	III-59
	X
xref images	III-86

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

Indicate any additions or changes you would like to see in the index.

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional) _____ Date _____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:
Publications Department
Rational
1501 Salado Drive
Mountain View, CA 94043