

*Rational Rose 2000e*  
*Rose Extensibility*  
*User's Guide*

---

---

**Copyright © 1998-2000 Rational Software Corporation.  
All rights reserved.**

Part Number 800-023328-000

Revision 7.0, March 2000, (Software Release 2000e)

This document is subject to change without notice.

GOVERNMENT RIGHTS LEGEND: Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14, as applicable.

Rational, the Rational logo, Rational Rose, ClearCase, and Rational Unified Process are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries. All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies. Basic Script is a trademark of Summit Software, Inc.



## *Contents*

---

**List of Figures** xiii

**List of Tables** xv

**Preface** xvii

Prerequisites xvii

How This Manual Is Organized xviii

Online Help xviii

Online Manual xix

Related Documentation xix

File Names xix

**Chapter 1 Basic Extensibility Concepts 1**

Rational Rose Extensibility 1

The REI Model and Rational Rose Extensibility 1

Rational Rose Scripting 4

Rational Rose Automation 4

Rational Rose Add-In Manager 5

Default Properties and Property Sets 5

Rational Rose Extensibility Type Libraries 5

<b>Chapter 2</b>	<b>Customizing Rational Rose Menus</b>	<b>7</b>
	Extending Rational Rose Menus	7
	Customizing Rational Rose Main Menus	7
	Procedure	8
	Adding Entries to a Rational Rose Menu File	9
	Menu File Keywords	10
	Menu Actions	11
	Menu File Variables and Modifiers	12
	Syntax Rules for Rational Rose Menu File Entries	15
	Adding Scripts to a Rational Rose Menu	16
	Adding or Editing the Virtual Path for Scripts	17
	Sample Rational Rose Menu File	19
	Customizing Rational Rose Shortcut Menus	21
	Benefits	21
	Limitations	22
	Key Terms and Concepts	22
	Language-Dependent	22
	Language-Neutral	23
	Language Add-In	23
	Non-Language Add-In	24
	Behind the Scenes of Shortcut Menus...	24
	How Rational Rose Formats and Displays Shortcut Menu Items	24
	Shortcut Menu Scenarios	26
	Shortcut Menu Design Considerations	28
	Procedure	29
	Creating Events for Shortcut Menus	30
	OnActivate	30
	OnEnableContextMenuItems	30
	OnSelectedContextMenuItem	31
	Adding Menu Items to the Shortcut Menu	31
	Working with Shortcut Menu Items	31
	Working with the Shortcut Menu Item Collection	31
	Editing Shortcut Menu Items	31
	Changing the State of a Shortcut Menu Item	32
	Sample Shortcut Menu Implementation Code	32
	Sample Rational Rose Script Shortcut Menu Code	34

---

## **Chapter 3 Using the REI to Work with Rational Rose 39**

Introduction	39
Getting the Rational Rose Application Object	39
Using Rational Rose Script	39
Using Rational Rose Automation	40
Associating Files and URLs with Classes	40
Managing Default Properties	41
Adding a Property to a Set	42
How To	42
Example	43
Notes on the Example	43
Creating a New Property	44
How To	44
Example	44
Notes on the Example	44
Deleting Model Properties	44
Getting Model Properties	45
Setting Model Properties	45
Setting Model Properties Using OverrideProperty	45
How To	45
Example	46
Notes on the Example	46
Setting Model Properties Using InheritProperty	46
How To	46
Example	47
Notes on the Example	47
Creating a New Property Set	47
Cloning a Property Set	47
How To	47
Example	48
Notes on the Example	48

Deleting a Property Set	49
How To	49
Example	49
Notes on the Example	49
Getting and Setting the Current Property Set	49
How To	49
Example	50
Notes on the Example	50
Creating a User-Defined Property Type	51
How To	51
Example	52
Notes on the Example	52
Creating a New Tool	53
Placing Classes in Categories	53
Using Type Libraries for Rational Rose Automation	53
How To	53
Example	53
Working with Controllable Units	54
Working with Rational Rose Diagrams	54
Getting an Element from a Collection	55
Accessing Collection Elements By Count	55
How To	55
Example	55
Accessing Collection Elements By Unique ID	56
How To	56
Example	56
Accessing Collection Elements By Name	56
How To	56
Example	56

---

## **Chapter 4 Using the Rational Rose Script Editor 57**

The Script Editor Window 57

Opening a Script 58

Creating New Rational Rose Scripts 58

    Creating a New Script from Scratch 58

    Creating a New Script from an Existing Script 59

Moving the Insertion Point in a Script 59

    Moving the Insertion Point with the Mouse 59

    Moving the Insertion Point to a Specified Line in Your Script 60

Selecting Text 60

    Selecting Text with the Mouse 61

    Selecting Text with the Keyboard 61

    Selecting an Entire Line 62

Deleting, Cutting, Copying, and Pasting Text 62

    Deleting Text 62

    Cutting a Selection 62

    Copying a Selection 62

    Pasting the Contents of the Clipboard into Your Script 63

Adding Comments to a Script 63

    Adding a Full-Line Comment 63

    Adding a Comment at the End of a Line of Code 63

Finding and Replacing Text 64

    Finding Specified Text 64

    Replacing Specified Text 65

Running, Pausing, and Stopping Your Script 66

    Running Your Script 66

    Pausing an Executing Script 66

    Stopping an Executing Script 66

Tracing Script Execution 67

    Stepping Through Your Script 67

    Displaying the Calls Dialog Box 68

Setting and Removing Breakpoints	69
Starting Debugging Partway through a Script	69
Continuing Debugging at a Line Outside the Current Subroutine	69
Debugging Selected Portions of Your Script	70
Removing a Single Breakpoint Manually	70
Removing All Breakpoints Manually	70
Working with Watch Variables	71
Adding Watch Variables	71
Selecting Variables on the Watch List	72
Deleting Watch Variables	73
Modifying the Value of Variables on the Watch Variable List	73
Compiling Your Script	74
Using Interscript Calls	74
Guidelines for Using a Script to Call Another Script	74
Debugging Interscript Calls	75
Working with the Dialog Editor	75
Inserting a Dialog Box into Your Script	75
Editing an Existing Dialog Box	75
Displaying and Adjusting the Grid	76
Changing Titles and Labels	77
Assigning Accelerator Keys	78
Capturing Standard Windows Dialog Boxes	78
Testing Your Dialog Boxes	79
Incorporating Dialog Boxes or Controls into Your Script	80
Selecting Controls	81
Selecting Dialog Boxes	81
Repositioning Items	82
Repositioning Items with the Mouse	82
Repositioning Items with the Arrow Keys	82
Repositioning Dialog Boxes with the Dialog Information Dialog Box	83
Repositioning Controls with the Dialog Information Dialog Box	83



---

Resizing Items	84
Resizing Items with the Mouse	84
Resizing Items with the Information Dialog Box	84
Resizing Selected Items Automatically	84
Adding Controls	85
Duplicating Controls	86
Adding Pictures to a Dialog	86
Adding Pictures from Files	86
Adding Pictures from Picture Libraries	87
Pasting Items into Dialog Editor	87
Pasting Existing Dialog Boxes into Dialog Editor	87
Pasting Controls from Existing Dialog Boxes into Dialog Editor	88
Displaying the Information Dialogs	88
Displaying the Information Dialog Boxes for Dialogs	88
Attributes You Can Adjust with the Dialog Box Information Dialog Box	89
Displaying the Information Dialog Boxes for Controls	90
Attributes You Can Adjust with the Information Dialog Boxes for Controls	90

## **Appendix A Rational Rose Script Editor Shortcuts 95**

General Shortcuts	95
Navigating Shortcuts	96
Editing Shortcuts	97
Debugging Shortcuts	98
File Menu Shortcuts	98
Edit Menu Shortcuts	99
Debugger Menu Shortcuts	100

### **Appendix B Developing Add-Ins for Rational Rose 101**

Introduction	101
Why Create Add-Ins?	103
Types of Add-Ins	103
What is in an add-in?	103
Main menus	104
Shortcut menu	104
Custom Specifications	104
Properties	105
Data types	105
Stereotypes	105
Online help	105
Context-sensitive help	105
Registering for events	105
Functionality	106
UNIX versus Windows	106
Creating portable add-ins	107
How to develop add-ins	108
Customizing Main Menus	109
Customizing the Shortcut Menu	110
Creating Custom Specifications	110
Customizing Properties	110
Design Considerations	110
Information in Property Files	111
Format for Property Files	112
Sample Property File	117
Creating Property Files	119
Testing Property Files	119
Customizing Data types	120
Customizing Stereotypes	120
Steps for Creating Add-In Stereotypes	121
Additional online help	126
Adding Online Help for Your Add-In	127

---

Additional context-sensitive help	128
Main Menu Items	128
Model Properties	129
User manuals	129
Registering for events	129
Interface Versus Script Events	130
What events are available?	130
How to add events to your add-in	131
Updating the Registry	134
Registry Entries	134
Registering Custom Stereotypes	137
Updating the registry during installation	137
Registry File Anatomy	138
Installing, Setting up, and Uninstalling your add-in	139
Installation Reminders	139
Installing Add-Ins	140
Uninstalling Add-Ins	140
Activating and deactivating add-ins	141

**Index 143**





## *List of Figures*

---

- Figure 1 Rational Rose Extensibility Model — Logical View 2
- Figure 2 Rational Rose Application and Extensibility Components 3
- Figure 3 Adding Virtual Path for Scripts 18
- Figure 4 Sample Rational Rose Menu File 19
- Figure 5 Sample Code for Shortcut Menus 34
- Figure 6 Property Specification Editor 41
- Figure 7 Rational Rose Script Editor 58
- Figure 8 Goto Line Dialog Box 60
- Figure 9 Selected Script Text 61
- Figure 10 Find Script Text Dialog Box 64
- Figure 11 Replace Dialog Box 65
- Figure 12 Script Calls Dialog Box 68
- Figure 13 Add Watch Dialog Box 71
- Figure 14 Modify Variable Dialog Box 73
- Figure 15 Grid Dialog Box 76
- Figure 16 Dialog Editor with Grid Displayed 77
- Figure 17 Capturing a Dialog Box 78
- Figure 18 Sample Dialog Box in Basic Script 81
- Figure 19 Dialog Box Information Dialog Box 89
- Figure 20 Control Information Dialog Box 90
- Figure 21 Rational Rose Add-Ins Architecture 102
- Figure 22 Sample Custom Properties 117
- Figure 23 OLEServer Windows Registry Entry 131
- Figure 24 Windows Registry Entries for Rational Rose Events 132

## List of Figures

---

- Figure 25 Sample Event Handler Defining an Add-In's Interfaces for Rational Rose Events 133
- Figure 26 Windows Registry Entries for an Add-In 134
- Figure 27 Windows Registry Entry for an Add-In's Custom Stereotype Configuration File 137



## *List of Tables*

---

Table 1	Menu File Keywords	10
Table 2	Menu Actions	11
Table 3	Menu File Variables	13
Table 4	Menu File Modifiers	13
Table 5	Displaying Shortcut Menu Items	26
Table 6	Sample Watch Expressions	72
Table 7	General Shortcuts	95
Table 8	Navigating Shortcuts	96
Table 9	Editing Shortcuts	97
Table 10	Debugging Shortcuts	98
Table 11	File Menu Shortcuts	98
Table 12	Edit Menu Shortcuts	99
Table 13	Debugger Menu Shortcuts	100
Table 14	UNIX versus Windows	106
Table 15	Property file data types	115







## Preface

---

The *Rational Rose 2000e, Rose Extensibility User's Guide* describes the Rational Rose Extensibility Interface (REI) and provides procedures for:

- Customizing (extending) Rational Rose menus
- Working with Rational Rose using the REI
- Working with the Rational Rose Script Editor, which is the scripting environment for working with the REI

## Prerequisites

---

This manual assumes that you are familiar with the Windows 95, Windows 98, or Windows NT 4.0 operating environment, object oriented design concepts, and how to use Rational Rose.

If you are unfamiliar with Rational Rose or object oriented design concepts, you should refer to the *Rational Rose 2000e, Using Rose* manual, as well as run the Rational Rose tutorial, which is included on your product CD.

Also note that you may need to adapt the syntax listed for each REI property and method to your particular programming language. If the listed syntax, does not meet your needs, consult your programming environment's help, programming language books, and outside documentation on the subject.

### How This Manual Is Organized

---

This manual contains the following four chapters and two appendixes:

- **Chapter 1**—Basic Extensibility Concepts  
Provides an overview of Rational Rose extensibility concepts.
- **Chapter 2**—Customizing Rational Rose Menus  
Provides syntax, examples, and procedures for adding submenus and menu commands to Rational Rose main and shortcut menus. Also, describes how Rational Rose formats and displays shortcut menu items.
- **Chapter 3**—Using the REI to Work with Rational Rose  
Tells how to perform many common Rational Rose tasks by using the Rational Rose Extensibility Interface, rather than the user interface.
- **Chapter 4**—Using the Rational Rose Script Editor  
Provides detailed instructions for working in the Rational Rose Script Editor, your environment for creating, debugging and running Rational Rose Script.
- **Appendix A**—Rational Rose Script Editor ShortCuts  
Lists the shortcuts available when working in the Rational Rose Script editor.
- **Appendix B**—Developing Add-Ins for Rational Rose  
Provides additional information for users who want to explore the use of add-ins. Tells how to combine Rational Rose customizations and automations into one package.

### Online Help

---

Rational Rose includes comprehensive online help with hypertext links and a two-level index. The online help includes all of the information found in this guide, as well as all of the information contained in the *Rational Rose 2000e, Rose Extensibility Reference*.

## Online Manual

---

Rational Rose includes all the user manuals online. Please refer to the `Readme.txt` file (found in the Rational Rose installation directory) for more information.

## Related Documentation

---

After installation and before you begin using Rational Rose and the Extensibility Interface, please review any `readme.txt` files and **Release Notes** to ensure that you have the latest information about the product. The release notes are included with your product documentation and are available online from the **Start** menu. The release notes also list the new and updated classes, properties, and methods. This information allows existing users to quickly discover what has changed since the last version of Rational Rose.

For additional resources, refer to the *Using Rational Rose* guide and online help. If you are new to Rational Rose, visual modeling, or the Unified Modeling Language (UML), you may also want to read the book, *Visual Modeling with Rational Rose and UML*, included with your product documentation.

## File Names

---

Where file names appear in examples, Windows syntax is depicted. To obtain a legal UNIX file name, eliminate any drive prefix and change the backslashes to slashes:

`c:\project\username`

becomes

`/project/username`





## Chapter 1

# *Basic Extensibility Concepts*

---

## **Rational Rose Extensibility**

---

Rational Rose provides several ways for you to extend and customize its capabilities to meet your specific software development needs. You can:

- Customize Rational Rose menus.
- Automate manual Rational Rose functions with Rational Rose Scripts (for example, diagram and class creation, model updates, document generation, etc.).
- Execute Rational Rose functions from within another application by using the Rational Rose Automation object (RoseApp).
- Access Rational Rose classes, properties and methods right within your software development environment by including the Rational Rose Extensibility Type Library in your environment.
- Activate Rational Rose add-ins using the Add-In Manager.

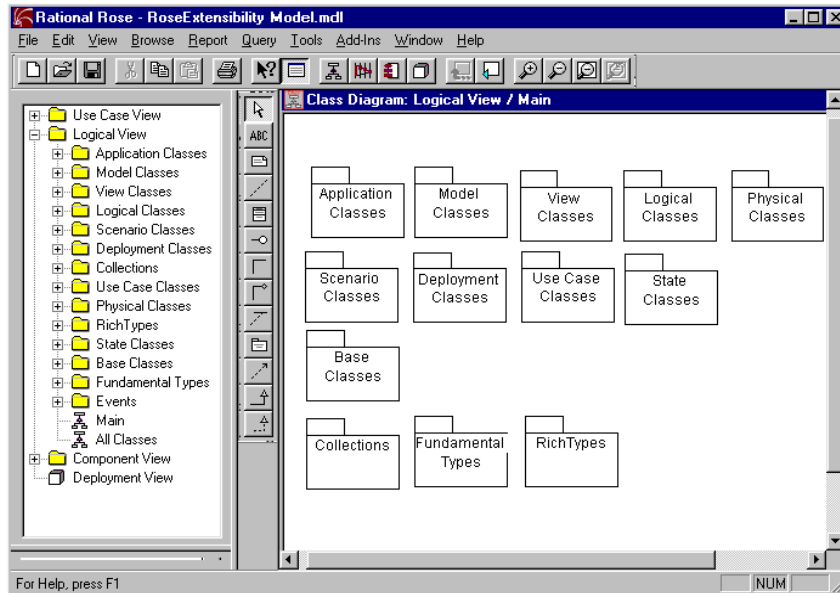
## **The REI Model and Rational Rose Extensibility**

---

The purpose of Rational Rose is to enable component based software development. As you would expect, the Rational Rose application is itself component-based, and is defined in the Rational Rose Extensibility Interface (REI) Model.

The REI Model is essentially a metamodel of a Rational Rose model, exposing the packages, classes, properties and methods that define and control the Rational Rose application and all of its functions.

Figure 1 shows the logical packages that comprise the Rational Rose Extensibility Interface Model. Refer to the *Rational Rose Extensibility Interface Reference* or online help for details on the classes contained in each package, and the properties and methods defined for each class.



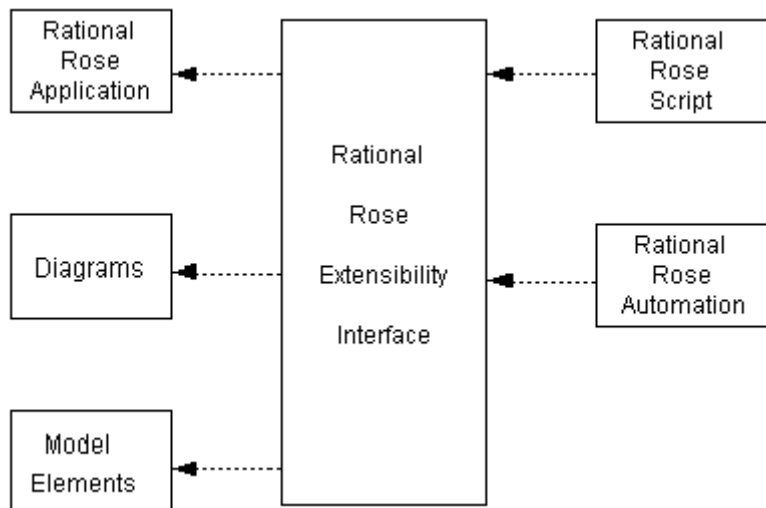
**Figure 1 Rational Rose Extensibility Model — Logical View**

You communicate with the Rational Rose Extensibility Interface through Rational Rose Scripts or through Rational Rose Automation. In either case, you will use the REI calls defined in the Rational Rose Extensibility Model and described in the *Rational Rose Extensibility Reference* and online help.

Figure 2 shows the components of Rational Rose and the Rational Rose extensibility interface, and illustrates the relationships between them. These components are:

- **Rational Rose Application**  
The Rational Rose Extensibility objects that interface to Rational Rose's application functionality.

- Rational Rose Extensibility Interface  
This is the common set of interfaces used by Rational Rose Script and Rational Rose Automation to access Rational Rose.
- Rational Rose Script  
The set of Rational Rose Script objects that allow Rational Rose Scripts to automate Rational Rose Functionality.
- Rational Rose Automation  
The set of Rational Rose Automation objects that allow Rational Rose to function as an OLE automation controller or server.
- Diagrams  
The Rational Rose Extensibility objects that interface to Rational Rose's diagrams and views.
- Model Elements  
The Rational Rose Extensibility objects that interface to Rational Rose's model elements.



**Figure 2 Rational Rose Application and Extensibility Components**

## Rational Rose Scripting

---

The Rational Rose Scripting language is an extended version of the Summit BasicScript language. The Rational Rose extensions allow you to automate Rational Rose-specific functions, and in some cases, perform functions that are not available through the Rational Rose user interface.

The Rational Rose script editor runs in the Rational Rose environment and provides access to the scripting environment. Start the script editor by clicking either **Tools > New Script** or **Tools > Open Script**.

Rational Rose provides a set of sample scripts that you can use as a base from which to create your own scripts.

- Check the **Scripts** folder in your Rational Rose installation directory for the complete list of available scripts.
- Use the Rational Rose Script Editor (click **Tools > Open Script**) to view a sample script. If you want to edit the script, click **File > Save Script As** to create a copy for your own use, leaving the sample intact.

Use the online BasicScript and Rational Rose Script Language References for complete script language information.

## Rational Rose Automation

---

Rational Rose automation allows you to integrate other applications with Rational Rose in two ways:

- Using Rational Rose as an automation controller, you can call an OLE automation object from within a Rational Rose script. For example, a Rational Rose script can use OLE automation to execute functions in applications such as Word and Visual Basic.
- Using Rational Rose as an automation server, you can call its OLE automation object from within other OLE-compliant applications.

Rational Rose Automation is accessible to automation controller environments, such as Visual Basic, EXCEL, Summit BasicScript, Softbridge Basic Language, C, C++, and others.



## Rational Rose Add-In Manager

---

The Rational Rose Add-In Manager provides you with the facilities required to install extensions you create as add-in components in the Rational Rose Environment.

In the extensibility environment, you can manipulate add-ins using calls to the `RoseAddInManager` object.

## Default Properties and Property Sets

---

Each Rational Rose model has its own default properties. These default properties are defined in a property file and are grouped into sets based on:

- Type of model element  
Class, component, relation, attributes, operations; and so on; the objects that make up the model.
- Tool  
Corresponds to a tab in the property specification; a tool can be a programming language tool, such as Java or C++; a database tool, such as Oracle8; a user-defined add-in to Rational Rose, or some other tool.
- Properties  
The actual properties and property values defined in the set; these must be appropriate to the model element and tool for which they are being defined.

**Note:** You can define multiple sets of default properties for the same tool and model element. For example, you might want one set of properties for a class with a stereotype of Actor and a different set of properties for a class with a stereotype of Interface. Both of these sets are still considered default properties in that they are predefined for the model. Defining multiple sets saves you work by minimizing the need to override properties later.

## Rational Rose Extensibility Type Libraries

---

Loading a type library for Rational Rose automation allows you to use Rational Rose class names to access the Rational Rose Extensibility Interface from your programming environment.

For example, if you are working in Visual Basic, instead of using the Basic object type **Object**, you can use the name of the actual Rational Rose class. You can also check the syntax of the properties and methods at compile time (early binding) instead of when the code is executed (late binding).

If you are working in Visual C, you can import **RationalRose.tlb** into an MFC project. This generates **ColeDispatchDriver** subclasses for each REI class, and methods allowing access to REI properties and methods.

**Important:** When you specify a Rational Rose class name in an automation environment, you must add the prefix **Rose** to the class name, unless the class name itself contains the word Rose already. (For example, the Rational Rose class, **RoseItem**, does not require a prefix.) This prefix prevents class name conflicts across applications.

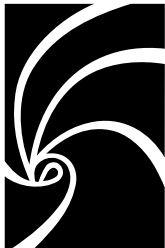
For example, in Rational Rose Script, the syntax for declaring a Category variable is:

```
Dim theCategory As Category
```

In Rational Rose Automation, the syntax for declaring a Category variable is:

```
Dim theCategory As RoseCategory
```

For details on using type libraries in any automation environment, refer to the documentation for your particular programming environment.



## *Chapter 2*

# *Customizing Rational Rose Menus*

---

## **Extending Rational Rose Menus**

---

Using the Rational Rose Extensibility Interface, you may add your own menu options to one of Rational Rose's menus (for example, **File**, **Edit**). You can also add your own menu options to the Rational Rose shortcut menu (displayed when you right-click).

This chapter explains how to customize:

- Rational Rose main menus
- Rational Rose shortcut menu

## **Customizing Rational Rose Main Menus**

---

You can extend or customize Rational Rose menus by updating the Rational Rose menu file, which Rational Rose reads during startup.

You can extend Rational Rose menus by adding:

- Submenus
- Menu options that execute any of the following:
  - Rational Rose primitives
  - Rational Rose scripts

- System commands
- External programs
- Menu separators (lines between menu options, used to group similar menu items)

**Note:** You can add information to existing menus (for example, **File**, **Edit**, etc.); however, you cannot add new menus to the Rational Rose menu bar.

The content of Rational Rose menus is defined in the `Rose.mnu` file. If you want to customize Rational Rose menus, you must edit this file.

While you cannot add new menus to the Rational Rose menu bar, you can add commands to the existing Rational Rose menus.

Use the procedures, commands, and syntax described in this chapter to add Rational Rose menu commands that:

- Execute a program or shell script
- Execute a Rational Rose script
- Load or save controllable units
- Display a dialog for user input
- Change write protection for a controllable unit
- Execute an interface in a COM server (for example, from your add-in)

### Procedure

The following procedure outlines the general steps for customizing Rational Rose menus.

The subsections following the procedure provide information on command syntax, variables, and modifiers to use as you complete the procedure.

Check the sample menu file at the end of this chapter for a complete example that illustrates how to put the various menu elements together into a working menu file.

1. Using any text editor, open the `Rose.mnu` file. (The file resides in the directory where Rational Rose is installed.)
2. Add entries to `Rose.mnu` for any or all of the following:
  - Submenus

- Menu options
- Menu separators

**Note:** Pay close attention to the syntax rules that apply to your entries to the Rational Rose menu file. For example, the syntax of the menu specifications includes opening and closing braces. You must include these braces in your specifications, or they will not work properly. For complete details, see *Syntax Rules for Rational Rose Menu File Entries*, later in this chapter.

3. If the menu item executes a script, add or edit Rational Rose's virtual path for scripts, unless one is already defined.
4. Save the file.
  - To create another menu file while leaving the `Rose.mnu` file intact, save the file under a different name. (Recommended)
  - To overwrite the file, save it as `Rose.mnu`

### Adding Entries to a Rational Rose Menu File

Using any text editor and the following information, you can add menu entries to the Rational Rose menu file. The entries will appear on the Rational Rose menu in the order in which you specify them.

As you add menu entries, you will specify:

- **Keywords** that determine what to add to the menu (a submenu, a menu option, a separator).
- **Menu actions** that specify what action to take when the menu item is selected.
- **Arguments** that further define a menu action, or that determine the conditions under which a menu command is enabled or disabled in Rational Rose.

Remember to follow all of the syntax rules as described in *Syntax Rules for Rational Rose Menu File Entries*, later in this chapter. For example, the syntax of the menu specifications includes opening and closing braces. You must include these braces in your specifications, or they will not work properly. Remember that each opening brace (`{`) requires a corresponding closing brace (`}`).

### Menu File Keywords

Table 1 describes the valid keywords for your entries in the Rational Rose menu file:

**Table 1** *Menu File Keywords*

---

<b>Keyword</b>	<b>Description</b>
<b>Menu</b> <i>RoseMenu</i>	Enter the <b>Menu</b> keyword, followed by the Rational Rose menu name to indicate the name of the menu being extended. For example, enter <code>Menu Tools</code> as the first line of an entry that extends the <b>Tools</b> menu.
<b>Menu</b> " <i>Menu Text</i> "	Enter the <b>Menu</b> keyword, followed by a text string to indicate the name of a submenu being added to the menu. Note that quotation marks are required if the text string contains spaces. For example, enter <code>Menu "My Scripts"</code> to add a submenu called <b>My Scripts</b> .
<b>Separator</b>	Enter the <b>Separator</b> keyword to add a separator to a list of menu options. Remember the placement of the Separator keyword controls the placement of the separator line on the menu.
<b>Option</b> " <i>Command text</i> "	Enter the <b>Option</b> keyword, followed by a text string to indicate the name of the menu command being added to the menu. Note that quotation marks are required if the text string contains spaces. For example, enter <code>Option "Run My Script"</code> to add a menu command called <b>Run My Script</b> .

---

## Menu Actions

An action defines the result of activating a menu entry. The required arguments can be supplied as keywords, constants, variables, or variables with modifiers. Table 2 describes the valid menu actions for your entries in the Rational Rose menu file.

**Table 2** *Menu Actions*

Action	Result
<b>Block</b>	Displays a modal dialog with 'arg' as its prompt. Used following 'exec' and an action such as the <b>Roseload</b> command to suspend the following action until the user chooses to continue.
<b>Rosascript</b> <i>Script-Path-and-Name</i>	Executes a source or compiled image of a script. You can specify the script name without its extension. The <b>Rosascript</b> command will search for the source script first and execute it if found. If not found, it will search for and execute the compiled script.
<b>Exec</b> <i>program-name</i> [ <i>arg2</i> [ <i>arg3</i> ...[ <i>arg10</i> ]]]	Executes the program or shell script contained in the file designated by <i>program-name</i> . (If the program is not located in the current directory, it must be in a directory in the execute path.) If the final argument is of the form '-F<filename>' then a file named <filename> is created (if it does not already exist). All arguments, except the last one are written to the file, and <filename> is passed as the sole argument to 'program.' <b>Note:</b> <ul style="list-style-type: none"> <li>■ F must be uppercase.</li> <li>■ It is up to 'program' to delete the file.</li> <li>■ To pass a string beginning with '-F' as the final parameter of an exec action, use '--F' instead. (The character '^' does NOT work in this case.)</li> </ul>
<b>Roseload</b> <i>ControlledUnit</i>	Loads the designated controlled unit(s) from the associated file.
<b>Rossave</b> <i>ControlledUnit</i>	Saves the designated controlled unit(s) to the associated file.

<b>Action</b>	<b>Result</b>
<b>Updateaccess</b> <i>ControlledUnit</i>	Sets the write protection for the controlled unit(s) to that of their corresponding files.
<b>InterfaceEvent</b> <i>add-in interface</i>	Executes the specified interface in the specified add-in's registered COM object. You are not limited to Rational Rose events. You may specify custom interfaces from your OLE server. Note that quotation marks are required if the add-in name contains spaces. <b>Examples:</b> <ul style="list-style-type: none"><li>■ <b>InterfaceEvent C++ OnBrowseHeader</b> When the user selects the menu option corresponding to this menu action, Rational Rose executes the OnBrowseHeader method in the C++ OLE server.</li><li>■ <b>InterfaceEvent All OnBrowseHeader</b> When the user selects the menu option corresponding to this menu action, Rational Rose executes the OnBrowseHeader method in all active add-ins' OLE servers.</li><li>■ <b>InterfaceEvent "My AddIn" CheckFormat</b> When the user selects the menu option corresponding to this menu action, Rational Rose executes the add-in's custom CheckFormat method in the "My AddIn" OLE server.</li></ul>

---

### Menu File Variables and Modifiers

Rational Rose provides a set of variables that correspond to various Rational Rose model items. You can use these variables in conjunction with a set of modifiers to determine the conditions under which menu items are enabled or disabled, as well as to specify specific menu actions.

The format for specifying variables with modifiers is:

```
variable [:mod1 [:mod2 [... [:mod10]]]]
```



### Variables

Table 3 lists the set of variables that are valid for extending Rational Rose menus.

**Table 3 Menu File Variables**

Variable	Description
<code>%all_units</code>	List of controlled units in all models
<code>%current_diagram</code>	Name of the current diagram
<code>%true</code>	Boolean value <b>true</b>
<code>%false</code>	Boolean value <b>false</b>
<code>%model</code>	Name of the current model
<code>%selected_items</code>	List of model elements selected in the current diagram
<code>%selected_units</code>	List of controlled units selected in the current diagram
<code>%uname</code>	Use in place of <code>%selected_units:first:elide</code> See <b>Modifiers</b> for information on <i>first</i> and <i>elide</i> .
<code>%ufile</code>	Use in place of <code>%selected_units:first:file</code> See <b>Modifiers</b> for information on <i>first</i> and <i>file</i> .

### Modifiers

Table 4 lists the set of modifiers that are valid for use with variables to extend Rational Rose menus.

**Table 4 Menu File Modifiers**

Modifier	Description
<code>allfiles</code>	Applied to a unit or item name or a list of unit or item names; evaluates to a string which contains the list of the corresponding header and source file names.
<code>basename</code>	Applied to a path, evaluates to a string that contains the file name portion of the path.  Applied to a list of paths, evaluates to a string that contains a list of file names. Each file name is extracted from its corresponding path.

<b>Modifier</b>	<b>Description</b>
codefile	<p>Applied to a unit or item name or a list of unit or item names, does one of the following:</p> <ul style="list-style-type: none"><li>■ Evaluates to a string which contains the complete path of the codefile attribute associated with the unit.</li><li>■ Evaluates to a string containing the name of the controlled unit in which the item is located.</li></ul> <p>Applied to a list, evaluates to a string which contains the list of corresponding file names.</p>
directory	<p>Applied to a path which resolves to a file, evaluates to a string that contains the directory portion of the path.</p> <p>Applied to a path which resolves to a directory, evaluates to a string that contains that path—no modification is performed.</p> <p>Applied to a list of paths, evaluates to a string that contains a list of directories. Each directory is extracted from its corresponding path using the preceding rules.</p>
elide	<p>Applied to a unit or item name, evaluates to the first space-delimited word in the name.</p> <p>Applied to a list, equivalent to <code>&lt;list&gt;:first:elide</code>.</p>
empty	<p>Applied to a list; evaluates to a boolean which is TRUE if the list is empty.</p>
false	<p>Applied to a boolean, evaluates to a boolean which is the logical negation of its input.</p>
file	<p>Applied to a controlled unit name, evaluates to a string that contains the path of the file associated with (providing persistent storage for) that controlled unit.</p> <p>Applied to a list of controlled unit names, evaluates to a string that contains a list of paths using the preceding rule for each controlled unit name in the input list.</p>
first	<p>Applied to an empty list, evaluates to NULL.</p> <p>Applied to a non-empty list; evaluates to a string that contains the first element of the list.</p>

---

<b>Modifier</b>	<b>Description</b>
headerfile	<p>Applied to a unit or item name or a list of unit or item names, does one of the following:</p> <ul style="list-style-type: none"> <li>■ Evaluates to a string which contains the name of the item's associated header file.</li> <li>■ Evaluates to a string containing a list of corresponding header file names.</li> </ul>
home_unit	Applied to a model component name, evaluates to a string containing the name of the controlled unit in which the item is located.
multiple	Applied to a list of unit, item, or file names, evaluates to boolean which is TRUE if list has more than one element.
not	A synonym (and preferred method) for false.
sourcefile	<p>Applied to a unit or item name, or to a list of unit or item names, does one of the following:</p> <ul style="list-style-type: none"> <li>■ Evaluates to a string, which contains the name of the item's associated sourcefile.</li> <li>■ Evaluates to a string containing a list of corresponding sourcefile names.</li> </ul>
unary	Applied to a list, evaluates to a boolean, which is TRUE if the input list has exactly one element.
writeable	<p>Applied to a path resolving to a file, evaluates to a boolean, which is TRUE if the file is writeable</p> <p>Applied to a controlled unit name, evaluates to a boolean, which is TRUE if the controlled unit is writable</p>

## Syntax Rules for Rational Rose Menu File Entries

Follow these rules when specifying menu text:

- When a text string contains embedded spaces, enclose the string in double quotation marks.  
For example, "Run Script"
- When a text string has no embedded spaces (a single word, for example), enter the string without any quotation marks.  
For example, Validate

- When a text string that is not enclosed in quotes includes a special character, the special character could be misinterpreted as a variable. For this reason, you must precede any special characters (such as ^, ", or %) with an escape character. The escape character for all special characters is ^.

### Examples:

- **Option** Calculate^%

Creates a menu option whose text reads **Calculate %**.

- **exec** Notepad ^" "c:\my files\file.txt"^"

Creates a menu action that executes the following command line: notepad "c:\my files\file.txt".

Note the escape character followed by an additional set of quotation marks. One set of quotation marks is necessary because there is a space in my files. The second set, each of which is preceded by the ^ escape character, causes the actual command line to include the quotation marks as part of the command.

- To create a mnemonic for the menu, add an ampersand (&) before the menu text.

For example, "&Run Script"

Allows users to execute the menu item by pressing CTRL+R.

- Menu text can include variables and modifiers.

For example, Option "Validate "%model

Creates a menu option with the text **Validate MyCurrentModel** (assuming the current model is called MyCurrentModel).

See *Menu File Variables and Modifiers*, earlier in this chapter, for more information.

## Adding Scripts to a Rational Rose Menu

If you create a Rational Rose script that you will use over and over again, you may want to add it to a Rational Rose menu. For example, if you write a script to create a particular report based on the contents of a model, you will probably run that script periodically.

Follow these steps to add such scripts to a Rational Rose menu:

1. Open the Rational Rose Menu file, or create a new one to use in its place.
2. Edit the Path Map so that it includes a virtual script path. (See *Adding or Editing the Virtual Path for Scripts*, next in this chapter.)
3. Modify the Rational Rose menu file to add the script under the appropriate menu, being careful to follow all of the menu file syntax rules. To do this:
  - In the menu file, locate the menu specification that corresponds to the Rational Rose menu to which you want to add the script. Each menu specification is comprised of the **Menu** keyword followed by the name of a Rational Rose menu. For example, the **Tools** menu specification begins with `Menu Tools`.
  - Within the appropriate menu specification, add a menu option that specifies the text of the menu command that will run the script (for example, **Run Conversion Wizard**).
  - Enter a `RoseScript` menu action to cause the script to execute when a user selects the menu command.
4. Save the updated menu file.

### Adding or Editing the Virtual Path for Scripts

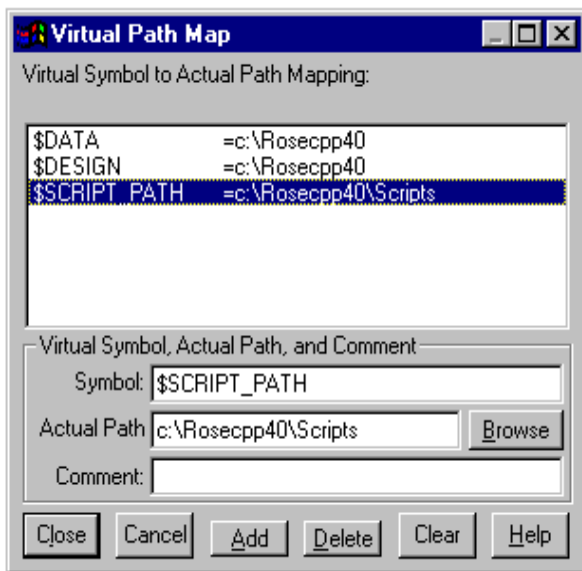
When you edit the Rational Rose menu file to include script commands, you must include one of the following:

- The fully qualified name of the script file to execute
- The virtual path that maps to the actual path

Defining a virtual path for scripts simplifies the process of editing the menu file by allowing you to specify the symbolic virtual path name instead of the complete file path.

Use the following procedure to add or edit a virtual path for scripts:

1. Start Rational Rose.
2. Click **File > Edit Path Map** to display the **Virtual Path Map** dialog box.



**Figure 3 Adding Virtual Path for Scripts**

3. Check for the **\$SCRIPT\_PATH** virtual symbol and do one of the following:
  - If the symbol exists, select it in the dialog to display its current mapping information in the lower portion of the dialog.
  - If the symbol does not exist, enter it in the **Symbol** field in the lower portion of the dialog.
  - Enter the actual path to your Rational Rose scripts, or use the **Browse** button to locate and select the path. (Normally these scripts reside in a **Scripts** subdirectory of the Rational Rose installation directory.)
  - When you make changes in the dialog, the **Close** button becomes an **OK** button. Click **OK** to save your changes and exit the **Virtual Path Map** dialog box.

## Sample Rational Rose Menu File

The following figure shows a portion of a Rational Rose menu file.

```

Rose.mnu - Notepad
File Edit Search Help
Menu Help
{
  Separator
  Menu "Rational on the &Web"
  {
    Option "&Online Support"
    {
      RoseScript $SCRIPT_PATH\webgorationalsupport.ebx
    }
    Option "Rational &Home Page"
    {
      RoseScript $SCRIPT_PATH\webgorational.ebx
    }
  }
}
Menu Report
{
option "Show &Participants in UC"
  {
    enable %selected_items:empty:false
    RoseScript $SCRIPT_PATH\participants.ebx
  }
option "&Documentation Report..."
  {
    RoseScript $SCRIPT_PATH\reportgen.ebx
  }
}
Menu Tools
{

```

**Figure 4** Sample Rational Rose Menu File

Note the following entries as you examine the menu specifications that comprise this file:

- Separator Entry

A separator entry causes a separator line to appear between the menu item above the keyword and the menu item below the keyword. In this case, a separator line will appear above the **Rational on the Web** submenu.

### ■ Menu Entry

A Rational Rose menu entry consists of the **Menu** keyword, followed by the name of the Rational Rose menu being extended.

This menu file extends the **Help** menu, the **Reports** menu, and the **Tools** menu (only partially in view).

### ■ Submenu Entry

A submenu is a second level menu that appears under a menu. A submenu entry looks just like a menu entry, with two exceptions:

- It appears within a Rational Rose menu specification. (In this case, it is part of the Rational Rose **Help** menu specification.)
- The **Menu** keyword is followed by the submenu title. (Notice that when the submenu title has embedded spaces, it is enclosed in quotation marks.)

### ■ Option Entry

Menu option entries define menu commands that you add to a menu. They begin with the **Option** keyword and are followed by the option title.

This file adds the following commands to the specified menus:

- Online Support (**Rational on the Web** submenu of the **Help** menu)
- Rational Home Page (**Rational on the Web** submenu of the **Help** menu)
- Show Participants in UC (**Report** menu)
- Documentation Report (**Report** menu)

### ■ Menu Action

Menu actions tell Rational Rose what to do when the menu item is selected. Each of the options in this file executes a Rational Rose Script. The Menu Actions topic describes all of the available actions.

### ■ Menu Argument

Menu arguments can be included to enable or disable a menu item under various circumstances. The arguments begin with either an **Enable** or **Disable** keyword, followed by variables and modifiers which define the circumstances.

In this case, the **Show Participants in UC** menu item will be enabled only if at least one item is selected in the current diagram (that is, the statement, “The list of selected items is empty” is false).



- Braces

Every menu specification entry must begin with a left brace ( { ) and end with a right brace ( } ). The nested braces allow you to define the hierarchy of a menu specification from the Rational Rose menu at the high end to a submenu option at the low end.

## Customizing Rational Rose Shortcut Menus

---

When you or the user of your add-in to Rational Rose right-clicks in Rational Rose, a shortcut menu appears. The commands displayed on the shortcut menu are determined by where you or your add-in user clicks the mouse and what items are selected in the diagram or browser. You can take advantage of this feature in your add-in's functionality so that your add-in user sees your shortcut menu items when they right-click. If your add-in has features that you want to include on a shortcut menu, the shortcut menu help topics explain how to add items to the Rational Rose shortcut menu by using the Rational Rose Extensibility Interface (REI).

### Benefits

The REI exposure of Rational Rose's shortcut menu interface provides the following benefits:

- Quicker access to your add-in's features for your customers
- Control of when the menu item displays on the shortcut menu
  - Default (any time the user selects multiple different items, e.g., classes and packages, or has nothing selected)
  - Diagram
  - Package
  - UseCase
  - Class
  - Attribute
  - Operation
  - Component
  - Role
  - Properties
  - Model

- DeploymentUnit
- ExternalDoc
- Control of the state in which your menu item displays on the shortcut menu (enabled, disabled, checked, unchecked)
- Control of the order (but not position) of multiple menu items on the shortcut menu
- Ability to add submenus to shortcut menu items
- Ability to add separator lines to the shortcut menu and submenus
- Ability to create one shortcut menu item that works for items selected in the browser as well as in a diagram (you do not have to create one menu item for items selected in the browser *and* another menu item for items selected in the diagram)

### Limitations

The position on the shortcut menu where your menu item displays is controlled by Rational Rose. If you have more than one item on the shortcut menu, however, you can control the order in which those items display by adding the items (using the **AddContextMenuItem** method) in the order in which you want the menu items displayed.

### Key Terms and Concepts

#### Language-Dependent

Rational Rose model elements are language-dependent if they can be associated with a specific language add-in, especially for code generation. These model elements are:

- Associations
- Attributes
- Classes
- Components
- Operations
- Roles

### Language-Neutral

Rational Rose model elements are language-neutral if they are not associated with a specific language. They are not generated into code (although model elements within them can be generated into code). These model elements are:

- Activities\*
- Decisions\*
- DeploymentUnits
- Diagrams\*
- ExternalDocs
- Models
- Packages\*
- Properties
- States\*
- Subsystems\*
- Swimlanes\*
- Synchronizations\*
- Transitions\*
- UseCases\*

\* These model elements make up a special subset of language-neutral items. A language add-in can add shortcut menu items for these model elements and get the `OnContextMenuItem` event for them, as long as the language add-in is set as the default language.

In other words, if your language add-in is the default language, when a user right-clicks on any of the above special language-neutral model elements, the user sees your language add-in's shortcut menu items for these model elements. If your language add-in is not the default language and the user right-clicks on one of these special language-neutral model elements, the user does not see your shortcut menu items for these model elements.

### Language Add-In

A language add-in is an add-in whose Rational Rose Registry setting for "LanguageAddIn" is set to Yes.

### Non-Language Add-In

A non-language add-in is an add-in whose Rational Rose Registry setting for “LanguageAddIn” is set to No.

### Behind the Scenes of Shortcut Menus...

When Rational Rose is started, it issues the **OnActivate** event and gets shortcut menu items (**ContextMenuItem**) from each add-in.

When you or the user of your add-in right-clicks, Rational Rose sends the **OnEnableContextMenuItems** event to the appropriate add-ins to get the applicable menu states for the add-in’s **ContextMenuItems**. Rational Rose then formats and displays the shortcut menu with appropriate add-in menu items depending on:

- Where the user right-clicked
- What items are selected (e.g., class, package)
- For what context the add-in’s shortcut menu items are defined (**ContextMenuItemType**)

See *How Rational Rose Formats and Displays Shortcut Menu Items*.

When the user selects a menu item from the shortcut menu, Rational Rose sends the **OnSelectedContextMenuItem** to the appropriate add-ins. It is then up to the add-in to map the event and arguments to one of its methods.

The add-in then runs the method that corresponds to the selected shortcut menu item.

### How Rational Rose Formats and Displays Shortcut Menu Items

The methodology for which shortcut menu items display and when they display makes the add-in a smart, seamless part of Rational Rose. Rational Rose only displays your shortcut menu when it is appropriate to do so. For example, Rational Rose displays your class shortcut items for that particular class if the following conditions are all true:

- Your add-in is a language add-in (defined by the registry setting)
- Your user creates a class with your language
- You have created the appropriate shortcut menu items (**ContextMenuItems**)

However, if your user creates a class with a different language add-in, Rational Rose does not display your class shortcut menu items. (A menu item for C++ classes might not make sense for a Visual Basic class.) Because of the flexibility that Rational Rose gives you through the REI to create shortcut menu items, it is also a complex concept. A shortcut menu item might not display when expected. It is, therefore, important to understand the scenarios (explained later in this chapter) for the complete explanation of what is displayed and when it is displayed.

Shortcut menu items created by a non-language add-in are always displayed on the appropriate menu. Those items created by a language add-in are displayed when the selected items have that language assignment. Shortcut menu items created by a Language add-in are also displayed when language-neutral items are selected and that language is the **Default Language**. Whether Rational Rose displays a particular shortcut menu item is dependent on the following considerations:

- The **Default Language** setting (click **Tools > Options > Notation**)
- Whether the selected items are the same type
- Whether the selected items are language-dependent or language-neutral
- Whether the selected items are associated with a particular Language add-in

See the Scenarios for examples of how these issues affect shortcut menu items.

## Shortcut Menu Scenarios

The following table describes all the possible scenarios for displaying shortcut menu items.

**Table 5 Displaying Shortcut Menu Items**

<b>Description</b>	<b>Example Selected Items</b>	<b>Displayed Shortcut Menu Items</b>
Same Language dependent types selected	Class 1 (Language A) (Any Default Language)	Language add-in A's <b>rsClass</b> shortcut menu items All non-language add-in's <b>rsClass</b> shortcut menu items
	Class 1 (Language A) Class 2 (Language A) (Any Default Language)	Language add-in A's <b>rsClass</b> shortcut menu items All non-language Add-in's <b>rsClass</b> shortcut menu items
	Class 1 (Language A) Class 2 (Language B) (Any Default Language)	Language add-in A's <b>rsClass</b> shortcut menu items Language add-in B's <b>rsClass</b> shortcut menu items All non-language Add-in's <b>rsClass</b> shortcut menu items
	Attribute 1 (Language A) Attribute 2 (Language B) (Any Default Language)	Language add-in A's <b>rsAttribute</b> shortcut menu items Language add-in B's <b>rsAttribute</b> shortcut menu items All non-language add-in's <b>rsAttribute</b> shortcut menu items
	Different language dependent types selected	Class 1 (Language A) Class 2 (Language B) Attribute 3 (Language A) Attribute 4 (Language C) (Any Default Language)
Different language dependent types selected	Operation 1 (Language A) Operation 2 (Language B) Role 3 (Language A) (Any Default Language)	Language add-in A's <b>rsDefault</b> shortcut menu items Language add-in B's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items

Description	Example Selected Items	Displayed Shortcut Menu Items
Nothing selected	(Default Language set to Language A)	Language add-in A's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items
	(Default Language set to Language B)	Language add-in B's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items
Same language neutral type selected	Diagram 1 (Created with any language) (Default Language set to Language A)	Language add-in A's <b>rsDiagram</b> shortcut menu items All non-language add-in's <b>rsDiagram</b> shortcut menu items
	Diagram 1 (Created with any language) (Default Language set to Language B)	Language add-in B's <b>rsDiagram</b> shortcut menu items All non-language add-in's <b>rsDiagram</b> shortcut menu items
	Diagram 1 (Created with any language) Diagram 2 (Created with any language) (Default Language set to Language A)	Language add-in A's <b>rsDiagram</b> shortcut menu items All non-language add-in's <b>rsDiagram</b> shortcut menu items
Same language neutral type selected (cont'd)	Diagram 1 (Created with any language) Diagram 2 (Created with any language) (Default Language set to Language B)	Language add-in B's <b>rsDiagram</b> shortcut menu items All non-language add-in's <b>rsDiagram</b> shortcut menu items
Different language neutral types selected	Diagram 1 (Created with any language) Package 2 (Created with any language) Package 3 (Created with any language) (Default Language set to Language A)	Language add-in A's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items

Description	Example Selected Items	Displayed Shortcut Menu Items
	Subsystem 1 (Created with any language) UseCase 2 (Created with any language) (Default Language set to Language B)	Language add-in B's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items
Combination of Language-dependent and Language-neutral types selected	Class 1 (Language A) Package 2 (Created with any language) (Default Language set to Language B)	Language add-in A's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items
	Class 1 (Language A) Class 2 (Language B) Package 3 (Created with any language) Package 4 (Created with any language) (Default Language set to Language C)	Language add-in A's <b>rsDefault</b> shortcut menu items Language add-in B's <b>rsDefault</b> shortcut menu items All non-language add-in's <b>rsDefault</b> shortcut menu items

**Note:** *If you want a shortcut menu item to display on more than one menu (for example, for classes and the default), you must create a separate **ContextMenuItem** for each item type (for example, one for **rsClass** and one for **rsDefault**). See the sample code later in this chapter.*

For more information on **rsClass**, **rsAttribute**, **rsDefault**, or **rsDiagram** see *ContextMenuItemType Enumeration* in the *Rational Rose 2000e, Rose Extensibility Reference*.

### Shortcut Menu Design Considerations

To keep the shortcut menu from becoming too cluttered with many different add-in menu options, try to keep menu items on the main shortcut menu to a minimum. Use submenus as much as possible. However, put all the important menu options on the main shortcut menu. Put less important menu options on a submenu under a generic main shortcut menu option.



Also, **Generate Code** and **Browse Code** menu options are no longer standard Rational Rose shortcut menu options. Each language add-in is now responsible for creating and manipulating these options according to their needs. This gives you greater control and flexibility with these features. When creating menu items, make the caption specific to your language (for example, Generate C++ Code, Generate Visual Basic Code). This reduces confusion since the user of your add-in could be using more than one language add-in in Rational Rose. Place **Generate Code** and **Browse Code** at the top level of the shortcut menu (as opposed to in a submenu).

You could also place shortcut menu items that open a custom specification sheet at the top level of the shortcut menu. However, if your add-in supports the **OnPropertySpecOpen** event, do not add a custom specification menu item because it would be redundant. This is due to the fact that when Rational Rose detects that the **OnPropertySpecOpen** event is supported for an item, Rational Rose adds the “Open Standard Specification” shortcut menu item (which displays the standard Rational Rose specification) immediately after the “Open Specification” shortcut menu item (which, in this context, displays the add-in’s custom specification sheet).

### Procedure

Follow these steps to customize the Rational Rose shortcut menu:

1. In order to use this feature of the REI, you must register your product in the Rational Rose Add-In Manager.
2. Determine the following:
  - What are your menu items?
  - Through what states will each menu item go?
  - Where should each menu item display (main shortcut menu or a submenu)?
  - In what order should your menu items display on the shortcut menu or submenu?
  - In which contexts should your menu options display (e.g., **rsDefault**, **rsClass**, **rsPackage**, etc.).
  - What circumstances will change menu states for each menu item?
  - Which access keys, if any, should you assign to each menu option?

- Are there any other considerations that are specific to your implementation?
- 3. Create the prototyped event methods, **OnActivate**, **OnEnableContextMenuItems**, and **OnSelectedContextMenuItem** customizing for your specific needs.
- 4. Create **ContextMenuItem** objects for each menu item by using the **AddContextMenuItem** method for each menu item. Use **AddContextMenuItem** in the order in which you want the menu items displayed on the shortcut menu.
- 5. Create your specific methods to support each **ContextMenuItem** (shortcut menu item) that maps to a specific function of your add-in. If the method already exists, update it as needed to take advantage of the Rational Rose shortcut menu.
- 6. Create and incorporate menu state changes as needed for your add-in. Use the **MenuState** property of the **ContextMenuItem** to change menu states.
- 7. Determine if there are any additional steps necessary for your specific implementation and perform those steps.

### Creating Events for Shortcut Menus

To customize the Rational Rose shortcut menu feature, your add-in must provide the following events:

#### OnActivate

Add shortcut menu items when your add-in gets the **OnActivate** event. If you do not already have this event in your add-in, you can customize it using the following prototype:

```
void OnActivate (LPDISPATCH pRoseApp)
```

#### OnEnableContextMenuItems

Provide an **OnEnableContextMenuItems** method in your add-in's OLE server with the following prototype:

```
Boolean OnEnableContextMenuItems (LPDISPATCH pRoseApp, VT_I2  
    itemType)
```

If your add-in does not provide this **OnEnableContextMenuItems** method, the system enables the shortcut menu items by default.

### OnSelectedContextMenuItem

Provide an **OnSelectedContextMenuItem** method in your add-in's OLE server with the following prototype:

```
Boolean OnSelectedContextMenuItem (LPDISPATCH pRoseApp, BSTR  
    internalName)
```

where *internalName* indicates the mapping from the selected shortcut menu item to the add-in's corresponding functionality.

If your add-in does not provide this event, your add-in cannot respond to a shortcut menu item selection.

### Adding Menu Items to the Shortcut Menu

To create and add menu items to the shortcut menu, use the **AddContextMenuItem** Method.

**Note:** *An add-in should add context menu items when it gets the **OnActivate** event.*

### Working with Shortcut Menu Items

When the user activates the shortcut menu with items selected in the browser or a diagram, Rational Rose sends the **OnEnableContextMenuItems** event to the specified language Add-In. The language add-in can then call **GetSelectedItems** at the model level to get all selected items, regardless of whether the user selected the items in the browser or in a diagram.

### Working with the Shortcut Menu Item Collection

To work with a subset or the set of all shortcut menu items, use the **GetContextMenuItems** Method.

The add-in can then iterate through the collection of **ContextMenuItems** by using the **GetAt** method, and setting the **MenuState** property accordingly.

### Editing Shortcut Menu Items

To change the properties of the shortcut menu item, see the **ContextMenuItem** class properties and methods.

### Changing the State of a Shortcut Menu Item

To enable, disable, check, or uncheck a particular shortcut menu item, change the **ContextMenuItem's MenuState** property.

### Sample Shortcut Menu Implementation Code

The following are sample pieces of code that you might use to add your menu items to Rational Rose's shortcut menu.

```
`Customize OnActivate from the prototype
Sub OnActivate (LPDISPATCH pRoseApp)
    . . .
    `Create all shortcut menu items
    Set myNewMenuItem1 =
        myAddIn.AddContextMenuItems (rsDefault, "Separator", "")
    Set myNewMenuItem2 = myAddIn.AddContextMenuItems (rsDefault,
        "Submenu &Main Add-In Menu Caption", "")
    Set myNewMenuItem3 = myAddIn.AddContextMenuItems (rsDefault,
        "&Caption 1", "internalName1")
    Set myNewMenuItem4 = myAddIn.AddContextMenuItems (rsDefault,
        "C&aption 2", "internalName2")
    Set myNewMenuItem5 = myAddIn.AddContextMenuItems (rsDefault,
        "endsubmenu", "")
    Set myNewMenuItem6 = myAddIn.AddContextMenuItems (rsDefault,
        "Separator", "")
    . . .
End Sub `OnActivate event

. . .
`Set initial state of each selectable shortcut menu item
myNewMenuItem2.MenuState = ENABLED
myNewMenuItem3.MenuState = ENABLED
myNewMenuItem4.MenuState = DISABLED
. . .

`Customize OnEnableContextMenuItems from the prototype
Function OnEnableContextMenuItems (LPDISPATCH pRoseApp, VT_I2
    itemType) As Boolean
    . . .
End Function `OnEnableContextMenuItems event

. . .
```

```
`Create each routine that corresponds to a selectable shortcut
`menu item
Sub DoMenuOption1 (argument1, argument 2, ...)
    . . .
End Sub `DoMenuOption1 subroutine

Function DoMenuOption2 (argument1, argument2, ...) As
    returnValue2
    . . .
End Function `DoMenuOption2 function

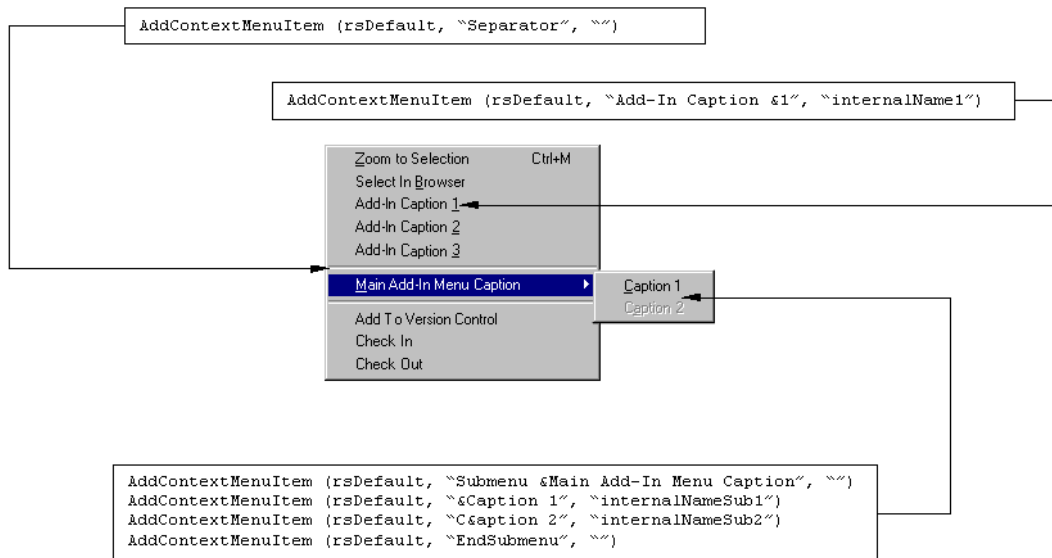
. . .

`Customize OnSelectedContextMenuItem from the prototype to map
`selectable shortcut menu items to functionality in this
` add-in.
Function OnSelectedContextMenuItem (LPDISPATCH pRoseApp, BSTR
    internalName) As Boolean
    Select Case internalName
        Case internalName1
            DoMenuOption1 (argument1, argument2, ...)
        Case internalName2
            x = DoMenuOption2 (argument1, argument2, ...)
    End Select `internalName
    . . .
End Function `OnSelectedContextMenuItem event

`Main program functionality
Sub Main
    . . .
End Sub `Main Program
```

## Sample Rational Rose Script Shortcut Menu Code

The sample RoseScript code below produced the following shortcut menu:



**Figure 5 Sample Code for Shortcut Menus**

```

'Subroutines to which the selectable shortcut menu items map
Sub internalName1

End Sub

Sub internalName2

End Sub

Sub internalName3

End Sub

Sub internalNameSub1

End Sub

Sub internalNameSub2
    
```

```
End Sub

Sub internalNameClass1

End Sub

Sub internalNameClass2

End Sub

Sub internalNameClass3

End Sub

Sub internalNameClassSub1

End Sub

Sub internalNameClassSub2

End Sub

Sub Main
    'Create a sample shortcut menu
    Dim myAddIn As RoseAddIn
    Dim myMenuItem As ContextMenuItem
    Dim myMenuItem2 As ContextMenuItem
    Dim myMenus As ContextMenuItemCollection
    Dim menuCount As Integer
    Dim i As Integer
    Dim classFound As Boolean
    Dim myItems As ItemCollection
    Dim itemCount As Integer
    Dim anItem As RoseItem
    Dim myModel As Model

    'ContextMenuItemType enumeration
    Const rsDefault As Integer = 0
    Const rsClass As Integer = 4

    'MenuState enumeration
    Const rsDisabled As Integer = 0
    Const rsEnabled As Integer = 1
```

```
Set myAddIn = ... `Get the add-in to which you want to add
`shortcut menu items.

`Create shortcut menu items for rsDefault
Set myMenuItem = myAddIn.AddContextMenuItem(rsDefault, "Add-
  In Caption &1", "internalName1")
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "Add-In Caption &2", "internalName2")
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "Add-In Caption &3", "internalName3")
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "Separator", "")
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "Submenu &Main Add-In Menu Caption", "")
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "&Caption 1", "internalNameSub1")
Set myMenuItem2 = myAddIn.AddContextMenuItem (rsDefault,
  "C&aption 2", "internalNameSub2")
Set myMenuItem2.MenuState = rsDisabled
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "endsubmenu", "")
Set myMenuItem = myAddIn.AddContextMenuItem (rsDefault,
  "Separator", "")

`Create exact same shortcut menu items for rsClass
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass, "Add-
  In Caption &1", "internalNameClass1")
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass, "Add-
  In Caption &2", "internalNameClass2")
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass, "Add-
  In Caption &3", "internalNameClass3")
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass,
  "Separator", "")
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass,
  "Submenu &Main Add-In Menu Caption", "")
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass,
  "&Caption 1", "internalNameClassSub1")
Set myMenuItem2 = myAddIn.AddContextMenuItem (rsClass,
  "C&aption 2", "internalNameClassSub2")
Set myMenuItem2.MenuState = rsDisabled
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass,
  "endsubmenu", "")
Set myMenuItem = myAddIn.AddContextMenuItem (rsClass,
  "Separator", "")
```



```
`Check to see if the user has selected only Class items. If
`so, enable the disabled shortcut menu option (Caption 2
`on the submenu).
classFound = True
Set myItems = RoseApp.CurrentModel.GetSelectedItems ()
itemCount = myItems.Count
For i = 1 To itemCount
    Set anItem = myItems.GetAt (i)
    If anItem.Stereotype <> "Class" Then
        classFound = False
    End If
Next i

If classFound = True Then
    myMenuItem2.MenuState = rsEnabled
End If

End Sub
```





## Chapter 3

# *Using the REI to Work with Rational Rose*

---

## **Introduction**

---

This chapter explains how to use the Rational Rose Extensibility Interface (REI) to accomplish many tasks that you would otherwise perform manually in the Rational Rose user interface.

This information is meant to orient you and to provide examples that you can use as starting points in your work with the REI.

The information in this chapter is not exhaustive. You should refer to the Extensibility Reference and Extensibility online help for complete descriptions of all of the REI classes, properties and methods. As you familiarize yourself with these, you will be able to realize the full capabilities that the REI makes available to you.

## **Getting the Rational Rose Application Object**

---

Whether you are using Rational Rose Script or Rational Rose Automation, you must get the Rational Rose Application object in order to control the Rational Rose application.

## **Using Rational Rose Script**

All Rational Rose Script programs have a global object called `RoseApp`, which has a property called `CurrentModel`. You must use `RoseApp.CurrentModel` to initialize the global Rational Rose object and subsequently open, control, save, or close a Rational Rose model from within a script.

The following sample code shows how to get the Rational Rose application object in a Rational Rose Scripting context:

```
Sub GenerateCode (theModel As Model)
    ' This generates code
End Sub

Sub Main
    GenerateCode RoseApp.CurrentModel
End Sub
```

### Using Rational Rose Automation

To use Rational Rose as an automation server, you must initialize an instance of a Rational Rose application object. You do this by calling either `CreateObject` or `GetObject` (or their equivalents) from within the application you are using as the OLE controller. These calls return the OLE Object which implements Rational Rose API's application object.

Refer to the documentation for the application you are using as OLE controller for details on calling OLE automation objects.

The following sample code shows how to get the Rational Rose application object in a Rational Rose Automation context:

```
Sub GenerateCode (theModel As Object)
    ' This generates code
End Sub

Sub Main
    Dim RoseApp As Object
    Set RoseApp = CreateObject ("Rose.Application")
    GenerateCode RoseApp.CurrentModel
End Sub
```

### Associating Files and URLs with Classes

---

Because Class objects inherit properties from `RoseItem`, you can define a set of external documents for any class. Each External Document has either a path property or a URL property.

- The Path property specifies a path to the file that contains the external document.
- The URL property specifies a Universal Resource Locator (URL) of a corresponding internet document.

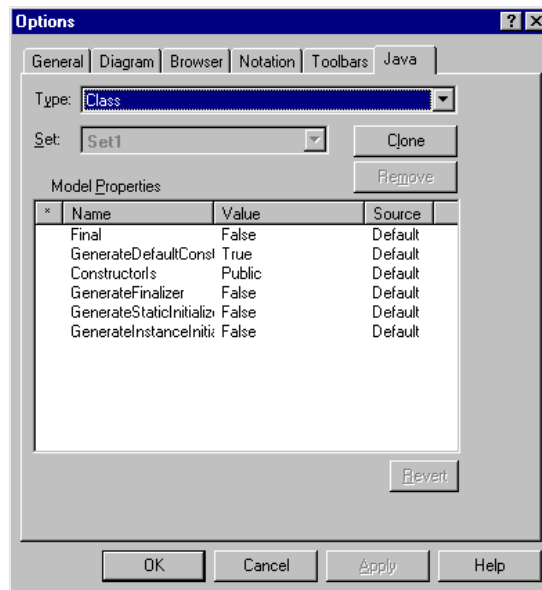
**Note:** Use *Extensibility online help* or refer to the *Extensibility Reference for syntax and other information*.

## Managing Default Properties

In the Rational Rose user interface environment, you manage a model's properties by using the specification editor.

To access the specification editor, click **Tools > Model Properties > Edit**.

You then select the appropriate tool tab, element type, and property set to edit. For example, in the following figure, the tool is **Java**, the model element type is **Class**, and the property set is **Set1**.



**Figure 6 Property Specification Editor**

From this point on, you can use the specification editor to edit individual properties, as well as clone (copy) and edit property sets. However, you cannot create new tools (tabs), new default property sets, or property types. For these capabilities, you must use the Rational Rose Extensibility Interface.

For more information on editing default properties and sets in the Rational Rose user interface, check the online help for information on specifications, or refer to the *Rational Rose 2000e, Using Rose* manual.

The next sections of this chapter explain how to work with properties and property sets in the extensibility environment.

In the Extensibility Interface, the `DefaultModelProperties` object manages the default model properties for the current model, and is itself a property of the model (expressed as `RoseApp.CurrentModel.DefaultProperties`). For this reason, default properties are applied to the current model only. When you create default properties they are applied and saved for the current model, but are not available to any new models you create.

To apply new properties to another model, re-run the script that creates the properties, specifying the new model as the current model

## Adding a Property to a Set

---

### How To

To add a property to a property set, define a subroutine that uses the `DefaultModelProperties.AddDefaultProperty` method. You will notice that this method requires you to pass 6 parameters:

- Class Name
- Tool Name
- Set Name
- Name of the New Property
- Property Type
- Value of the New Property

## Example

```
Sub AddDefaultProperties (theModel As Model)
  Dim DefaultProps As DefaultModelProperties
  Set DefaultProps = theModel.DefaultProperties
  myClass$ = theModel.RootCategory.GetPropertyClassName ()
  b = DefaultProps.AddDefaultProperty (myClass$,
    "ThisTool", "Set1", "StringProperty", "String", "")
  b = DefaultProps.AddDefaultProperty (myClass$,
    myTool$, "Set1", "IntegerProperty", "Integer", "0")
  b = DefaultProps.AddDefaultProperty (myClass$,
    myTool$, "Set1", "FloatProperty", "Float", "0")
  b = DefaultProps.AddDefaultProperty (myClass$,
    myTool$, "Set1", "CharProperty", "Char", " ")
  b = DefaultProps.AddDefaultProperty (myClass$,
    myTool$, "Set1", "BooleanProperty", "Boolean", "True")
End Sub
```

## Notes on the Example

1. When you specify the Class Name parameter, you must specify the internal name of the model element. There are two ways to obtain this information:
  - If properties are already defined for this element, it will appear in the specification dialog in the Rational Rose user interface. Simply check the specification editor and use the Type drop-down list to find the appropriate class name.
  - Use the `Element.GetPropertyClassName` method. This is the method used in the sample script. This example retrieves the internal name and returns it in `myClass$`, which is then passed as the class name parameter.
2. If the tool you specify does not exist, a new tool will be created. This is actually the only way to add a new tool to a model.
3. This example adds a property of each of the predefined property types (string, integer, float, char, boolean), with the exception of the enumeration type. You use the enumerated type to create your own property types and add enumerated properties to a set. See *Creating a User-Defined Property Type*, later in this chapter, for instructions and an example.

### Creating a New Property

---

#### How To

To create a new property that is not based on an existing property, use the `Element.CreateProperty` method. However, if you simply want to set an existing property to a different current value, you should use `Element.InheritProperty` or `Element.OverrideProperty` instead.

#### Example

```
' Property creation:  
b = theModel.RootCategory.CreateProperty (myTool,  
    "Saved", "True", "Boolean")  
b = theModel.RootCategory.InheritProperty (myTool, "Saved")
```

#### Notes on the Example

1. The `CreateProperty` call in the example creates a new property called **Saved**. It applies to the tool **MyTool**, its value is **True** and its type is **Boolean**.
2. The `InheritProperty` call in the example deletes the property just created. Because there is no default value to which such a property can return, `InheritProperty` effectively deletes it from the model.
3. For more information, see *Setting Model Properties Using InheritProperty*, *Setting Model Properties Using OverrideProperty*, and *Deleting Model Properties*, later in this chapter.

### Deleting Model Properties

---

If you are deleting a property that belongs to a property set, you can use the `DefaultModelProperties.DeleteDefaultProperty` method to delete the property from a model.

However, if you created a property using the `Element.CreateProperty` method, that property is not part of a property set. To delete such a property, use the `Element.InheritProperty` method.



## Getting Model Properties

---

The `Element` class provides two methods for retrieving information about model properties:

- To get the current value for a model property, whether inherited or overridden, use the `Element.GetPropertyValue` method. This method returns the value as a string.
- To retrieve the property object itself, use the `Element.FindProperty` method.

## Setting Model Properties

---

There are several ways to set model properties using the Extensibility Interface:

- Use the `Element.OverrideProperty` method to change only the value of a property, and keep all other aspects of the property definition intact.
- Use the `Element.InheritProperty` method to return a previously overridden property to its original value.
- Use the `Element.CreateProperty` or the `DefaultModelProperties.AddDefaultProperty` method to define a new property from scratch.

For more information, see *Creating a New Property*, earlier in this chapter.

### Setting Model Properties Using `OverrideProperty`

#### How To

The `Element.OverrideProperty` method allows you to use the default property definition and simply change its current value. Alternately, you could create a brand new property by calling the `Element.CreateProperty` method, but that would require you to specify the complete property definition, not just the new value.

If the property you specify does not exist in the model's default set, a new property is created for the specified object only. This new property is created as a string property.

### Example

```
Sub OverrideRadioProps (theCategory As Category)
    b = theCategory.OverrideProperty (myTool$, "StringProperty",
        "This string is overridden")
    b = theCategory.OverrideProperty (myTool$,
        "IntegerProperty", "1")
    b = theCategory.OverrideProperty (myTool$, "FloatProperty",
        "111.1")
    b = theCategory.OverrideProperty (myTool$,
        "EnumeratedProperty", "Value2")
End Sub
```

### Notes on the Example

1. Each of the 4 lines of the sample subroutine changes the current value of a specific property as follows:
  - The property called **StringProperty** now has a value of **This string is overridden**.
  - The property called **IntegerProperty** now has a value of 1.
  - The property called **FloatProperty** now has a value of 111.1.
  - The property called **EnumeratedProperty** now has a value of **Value2**.
2. Everything except for current value (tool name, class name, set, property name and property type) remains the same for the properties.

## Setting Model Properties Using InheritProperty

### How To

Use the `Element.InheritProperty` method to reset an overridden property to its original value.

You can also use this method to delete a property that you created using the `Element.CreateProperty` method. Because there is no default value to which such a property can return, `InheritProperty` effectively deletes it from the model.

### Example

```
Sub InheritRadioProps (theCategory As Category)
    b = theCategory.InheritProperty (myTool$, "StringProperty")
    b = theCategory.InheritProperty (myTool$, "IntegerProperty")
    b = theCategory.InheritProperty (myTool$, "FloatProperty")
    b = theCategory.InheritProperty (myTool$,
        "EnumeratedProperty")
End Sub
```

### Notes on the Example

Each of the 4 lines of the sample subroutine returns the current value of the specified property to its original value.

## Creating a New Property Set

---

To create a new property set from scratch, use the `DefaultModelProperties.CreateDefaultPropertySet` method.

## Cloning a Property Set

---

### How To

Cloning allows you to create a copy of an existing property set for the purpose of creating another property set. This is the easiest way to create a new property set, and is particularly useful for creating multiple sets of the same properties, but with different values specified for some or all of the properties.

To clone a property set in a model, use the `DefaultModelProperties.CloneDefaultPropertySet` method.

### Example

```
Sub CloneDefaultProperties (theModel As Model)
    Dim DefaultProps As DefaultModelProperties
    Set DefaultProps = theModel.DefaultProperties
    AddDefaultProperties theModel
    myClass$ = theModel.RootCategory.GetPropertyClassName ()
    b = DefaultProps.CloneDefaultPropertySet (myClass$, myTool$,
        "default", "SecondSet")
    b = DefaultProps.CloneDefaultPropertySet (myClass$, myTool$,
        "default", "ThirdSet")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "SecondSet", "StringProperty", "String", "Unique to
        SecondSet")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "SecondSet", "IntegerProperty", "Integer", "11")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "SecondSet", "FloatProperty", "Float", "89.9000")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "SecondSet", "EnumeratedProperty",
        "EnumerationDefinition", "Value2")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "ThirdSet", "StringProperty", "String", "Unique to
        ThirdSet")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "ThirdSet", "IntegerProperty", "Integer", "20")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "ThirdSet", "FloatProperty", "Float", "90.9000")
    b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
        "ThirdSet", "EnumeratedProperty",
        "EnumerationDefinition", "Value3")
End Sub
```

### Notes on the Example

1. This example clones an existing property set twice in order to define a total of three sets for the class and tool to which the sets apply.
2. All three sets have the same properties as those defined in the original set. In addition, several new properties are added to the second set and several other new properties are added to the third set.

## Deleting a Property Set

---

### How To

To delete an entire property set from a model, use the `DefaultModelProperties.DeleteDefaultPropertySet` method.

### Example

```
Sub DeleteDefaultProperties (theModel As Model)
    Dim DefaultProps As DefaultModelProperties
    Set DefaultProps = theModel.DefaultProperties
    myClass$ = theModel.RootCategory.GetPropertyClassName ()

    b = DefaultProps.DeleteDefaultPropertySet (myClass$,
        myTool$, "SecondSet")
    b = DefaultProps.DeleteDefaultPropertySet (myClass$,
        myTool$, "ThirdSet")

    b = theModel.RootCategory.SetCurrentPropertySetName
        (myTool$, "default")
End Sub
```

### Notes on the Example

1. The `Element.GetPropertyClassName` retrieves the valid internal class name to pass as a parameter on the delete calls.
2. Each `DefaultModelProperties.DeleteDefaultPropertySet` call deletes a property set from the model.
3. The `Element.SetCurrentPropertySetName` call sets the tool's current property set to its original set, which happens to be called **default**.

## Getting and Setting the Current Property Set

---

### How To

To find out which property set is the current set for a tool, use the `Element.GetCurrentPropertySetName` method.

To set the current property set to a particular set name, use the `Element.SetCurrentPropertySetName` and specify the set of your choice.

**Note:** When setting the current property set, you must supply a set name that is valid for the specified tool. To retrieve a list of valid set names for a tool, use `Element.GetDefaultSetNames`.

### Example

```
Sub RetrieveElementProperties (theElement As Element)
    Dim AllTools As StringCollection
    Dim theProperties As PropertyCollection
    Dim theProperty As Property
    Set AllTools = theElement.GetToolNames ()
    For ToolID = 1 To AllTools.Count
        ThisTool$ = AllTools.GetAt (ToolID)
        theSet$ = theElement.GetCurrentPropertySetName (ThisTool$)
        Set theProperties = theElement.GetToolProperties
            (ThisTool$)
        For PropID = 1 To theProperties.Count
            Set theProperty = theProperties.GetAt (PropID)
        Next PropID
    Next ToolID
End Sub
```

### Notes on the Example

1. `GetToolNames` retrieves the tool names that apply to the model element type called **Element** and returns them as a string collection called **AllTools**.
2. The current property set is retrieved for each tool name.
3. `GetToolProperties` retrieves the property collection that belongs to the current tool.
4. Each property that belongs to the tool's property collection is retrieved.

## Creating a User-Defined Property Type

---

### How To

Rational Rose Extensibility predefines the following set of property types:

- String
- Integer
- Float
- Char
- Boolean
- Enumeration

When you add properties to a set, you specify one of these types.

In addition, you can define your own property types and add properties of that type to a property set.

To create a user-defined property type, add a property whose type is enumeration and whose value is a string that defines the possible values for the enumeration.

Once you have defined the new type, adding a property of this new type is like adding any other type of property.

### Example

```
Sub AddDefaultProperties (theModel As Model)
    Dim DefaultProps As DefaultModelProperties
    Set DefaultProps = theModel.DefaultProperties
    myClass$ = theModel.RootCategory.GetPropertyClassName ()
    b = DefaultProps.AddDefaultProperty (myClass$, "myTool",
        "Set1", "MyNewEnumeration", "Enumeration",
        "Value1,Value2,Value3")
    b = DefaultProps.AddDefaultProperty (myClass$, "myTool",
        "Set1", "MyEnumeratedProperty", "MyNewEnumeration",
        "Value1")
    b = DefaultProps.AddDefaultProperty (myClass$, "myTool",
        "Set1", "isAppropriate", "Boolean", "True")
    b = DefaultProps.AddDefaultProperty (myClass$, "myTool",
        "Set1", "mySpace", "Integer", "5")
End Sub

Sub Main
    AddDefaultProperties (RoseApp.CurrentModel)
End Sub
```

### Notes on the Example

1. This example uses `Element.GetPropertyClassName` to retrieve the internal name of the class to which the property type will apply.
2. The first `AddDefaultProperty` call adds the enumeration and defines its possible values in the string "Value1, Value2, Value3."
3. The second `AddDefaultProperty` call adds a new property of the new enumerated type; the property value is set to "Value1."
4. If you want a new type to appear in the specification dialog in the Rational Rose user interface, you must actually add a property of that type to the set. Using the above example, if you simply created the type **MyNewEnumeration**, but did not add the property **MyEnumeratedProperty**, **MyNewEnumeration** would not appear in Type drop-down. Once you add the actual property, **MyNewEnumeration** would appear in the list of types.



## Creating a New Tool

---

There is no explicit way to add a new tool (tab) to a model. However, when you create a new property set or add a new property to a model, you must specify the tool to which the property or set applies. If the tool you specify does not already exist, it will be added during the create or add process.

## Placing Classes in Categories

---

- To create a new class and place it in a category, you use the `Category.AddClass` method.
- To relocate an existing class from one category to another, use the `Category.RelocateClass` method.

## Using Type Libraries for Rational Rose Automation

---

### How To

When you specify an REI class in an automation environment, you must add the prefix **Rose** to the class name, unless the word Rose is already part of the REI class name.

For more information on using Type Libraries with Rational Rose, see *Rational Rose Extensibility Type Libraries*, in Chapter 1 of this guide.

### Example

- In Rational Rose Script, the syntax for retrieving the Root Category of a model (that is, its logical view) is:

```
Model.RootCategory
```

- In Rational Rose Automation, the syntax for retrieving the Root Category of a model is:

```
RoseModel.RootCategory
```

- In both Rational Rose Script and Rational Rose Automation, the syntax for retrieving the documentation belonging to a RoseItem is:

```
RoseItem.Documentation
```

### Working with Controllable Units

---

Working with controllable units allows you to divide a model into smaller units. This is particularly useful for multi-user development, as well as for placing a model under configuration management.

The methods that apply to working with controllable units are:

- `ControllableUnit.Control` method, which associates a controllable unit with a file name, so that it can be passed to a configuration management application.
- `ControllableUnit.Uncontrol` method, which removes the file association from the unit.
- `ControllableUnit.Load` and `ControllableUnit.Unload` methods, which load or unload parts of a model (for example, the units for which a person is responsible).
- `ControllableUnit.Save` or `ControllableUnit.SaveAs` methods, which actually write the specified controllable unit to a file.

**Note:** When you save a model, that will also save its controllable units.

### Working with Rational Rose Diagrams

---

Each kind of Rational Rose diagram (class, component, scenario, etc.) inherits from the `Diagram` class.

A diagram is made up of `Items` and `ItemViews`. An `ItemView` is the physical representation of the actual Rational Rose item. As such, it is an object with properties and methods that define its appearance in the diagram window (position, color, size, etc). You can define multiple `ItemViews` for any given `RoselItem`.

- Use `Diagram.ItemViews` to iterate through the collection of item views belonging to a diagram.
- Use `Diagram.Items` to iterate through the items that exist in the diagram.
- Use `Diagram.GetViewFrom` to find the first itemview of a given item.

**Note:** You can only use `GetViewFrom` to retrieve the first itemview defined for the item. Even if you have more than one view, you'll always only get the first.

- To find out which itemviews are currently selected in a diagram, iterate through the diagram's itemviews. As you retrieve each itemview, use the `ItemView.IsSelected` method to find out whether it is currently selected in the diagram. You can then retrieve the selected itemview, or do any other processing you wish to do based on whether itemview is selected.
- A short way to retrieve all selected items from a diagram is to use the `Diagram.GetSelectedItems` method. Instead of iterating through the diagram and checking each itemview, this method simply returns everything that is selected.

## Getting an Element from a Collection

---

There are three ways to get an individual model element from a collection:

- Use the `GetwithUniqueID` method to directly access the element.
- Iterate through the collection using the element's name using `FindFirst`, `FindNext`, and `GetAt`.
- Iterate through the collection using `Count` followed by `GetAt`.

For more information, check the Extensibility Reference or online help for Collection Properties and Methods.

## Accessing Collection Elements By Count

### How To

Follow these steps to access collection elements by count:

1. Iterate through the collection using the `Count` property.
2. Retrieve the specific element using the `GetAt` method when the specific element is found.

### Example

```
Dim AllClasses As ClassCollection
Dim theClass As Class
For ClsID = 1 To AllClasses.Count
    Set theClass = AllClasses.GetAt (ClsID)
    ' ToDo: Add your code here...
Next ClsID
```

### Accessing Collection Elements By Unique ID

#### How To

The most direct and easiest way to get an element from within a collection is by unique ID. Follow these steps to access collection elements by unique ID:

1. Use the GetUniqueID method to obtain the element's unique ID.
2. Use the GetwithUniqueID method, specifying the ID you obtained in step 1.

#### Example

```
Dim theClasses As ElementCollection
Dim theClass As Element
theID = theClasses.theClass.GetUniqueID ()
theClass = theClass.GetwithUniqueID (theID)
```

### Accessing Collection Elements By Name

#### How To

Follow these steps to access an operation belonging to a class:

1. Use FindFirst to find the first occurrence of the specified operation in the collection
2. Use FindNext to iterate through subsequent occurrences of the operation
3. Retrieve the specific operation using the GetAt method when the specific operation is found

#### Example

```
Sub PrintOperations (theClass As Class, OperationName As
String)
Dim theOperation As Operation
OperID = theClass.Operations.FindFirst (OperationName$)
Do
Set theOperation = theClass.Operations.GetAt (OperID)
' ToDo: Add your code here...
OperID = theClass.Operations.FindNext (OperID,
OperationName$)
Loop Until OperID = 0
End Sub
```



## Chapter 4

# *Using the Rational Rose Script Editor*

---

The Rational Rose Script Editor provides your environment for creating, debugging, and compiling scripts that work with the Rational Rose Extensibility Interface.

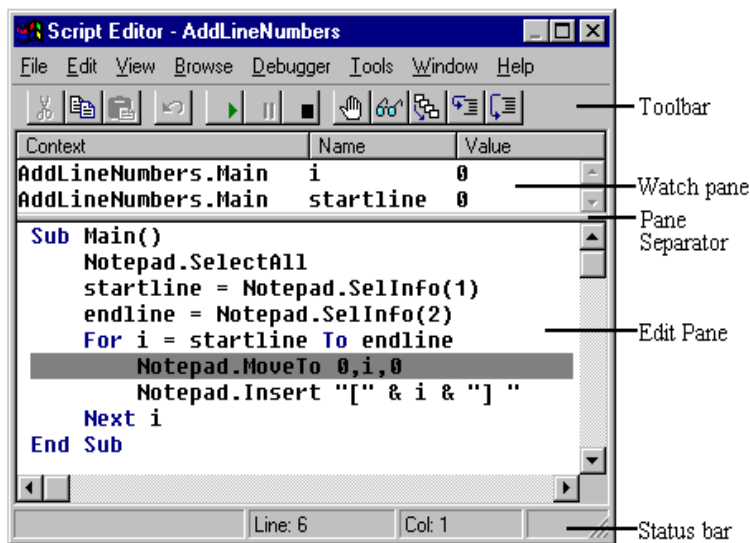
## **The Script Editor Window**

---

As shown in Figure 7, the Script Editor's application window contains the following elements:

- **Toolbar:** a collection of tools that you can use to provide instructions to the Script Editor
- **Edit pane:** a window containing the source code for the script you are currently editing
- **Watch pane:** a window that opens to display the watch variable list after you have added one or more variables to that list
- **Pane separator:** a divider that appears between the edit pane and the watch pane when the watch pane is open

- **Status bar:** displays the current location of the insertion point within your script



**Figure 7 Rational Rose Script Editor**

## Opening a Script

---

Use the following procedure to open a script in the Script Editor.

1. Click **Tools > Open Script**.
2. Select the script to open and select **OK**.

The script is displayed in a new Script Editor window.

## Creating New Rational Rose Scripts

---

### Creating a New Script from Scratch

Use the following procedure to create a new script in the Script Editor.

1. Click **Tools > New Script**.
2. Enter your script in the new Script Editor window.
3. Enter your script text.
4. Click **File > Save As** and save the new script.

### Creating a New Script from an Existing Script

Use the following procedure to create a new script from an existing script:

1. Click **Tools > Open Script**.
2. Select a file from the list of available scripts.
3. Click **OK** to enter the Script Editor and display the script.
4. Select the script text and click **Copy** to save the script text to the clipboard.
5. Click **Tools > New Script**.
6. Click **Paste** to paste the existing script text into the new script window.
7. Click **File > Save As** and save the new script.

### Moving the Insertion Point in a Script

---

There are two ways to move the insertion point in a script:

- With the mouse
- By specifying a line number

#### Moving the Insertion Point with the Mouse

Use the following procedure to use the mouse to reposition the insertion point. This approach is especially fast if the area of the screen to which you want to move the insertion point is currently visible.

1. Use the scroll bars at the right and bottom of the display to scroll the target area of the script into view if it is not already visible.
2. Place the mouse pointer where you want to position the insertion point.
3. Click the left mouse button.

The insertion point is repositioned.

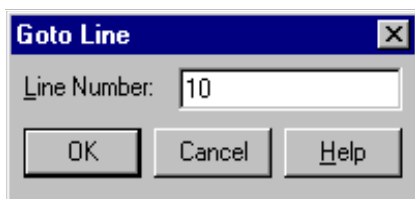
**Note:** *When you scroll the display with the mouse, the insertion point remains in its original position until you reposition it with a mouse click. If you attempt to perform an editing operation when the insertion point is not in view, the Script Editor automatically scrolls the insertion point into view before performing the operation.*

### Moving the Insertion Point to a Specified Line in Your Script

Use the following procedure to jump directly to a specified line in your script. This approach is especially fast if the area of the screen to which you want to move the insertion point is not currently visible but you know the number of the target line.

1. Click **Edit > Goto Line**.

The Script Editor displays the **Goto Line** dialog box.



**Figure 8 Goto Line Dialog Box**

2. Enter the number of the line in your script to which you want to move the insertion point.
3. Click **OK** button or press ENTER.
4. The insertion point is positioned at the start of the line you specified. If that line was not already displayed, the Script Editor scrolls it into view.

**Note:** *The insertion point cannot be moved so far below the end of a script as to scroll the script entirely off the display. When the last line of your script becomes the first line on your screen, the script will stop scrolling, and you will be unable to move the insertion point below the bottom of that screen.*

### Selecting Text

---

There are three ways to select text in an open script:

- With the mouse
- With the keyboard
- By selecting an entire line

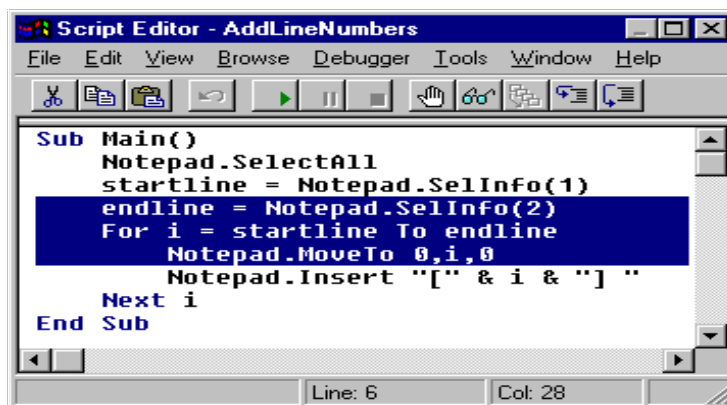


## Selecting Text with the Mouse

Use the following procedure to use the mouse to select text in your script.

1. Place the mouse pointer where you want your selection to begin.
2. Do one of the following:
  - While pressing the left mouse button, drag the mouse until you reach the end of your selection, and release the mouse button.
  - While pressing SHIFT, place the mouse pointer where you want your selection to end and click the left mouse button.

The selected text is highlighted on your display.



**Figure 9** Selected Script Text

## Selecting Text with the Keyboard

Use the following procedure to use keyboard shortcuts to select text in your script.

1. Place the insertion point where you want your selection to begin.
2. While pressing SHIFT, use one of the navigating keyboard shortcuts to extend the selection to the desired ending point.

The selected text is highlighted on your display.

### Selecting an Entire Line

Use the following procedure to use the keyboard to select one or more whole lines in your script.

1. Place the insertion point at the beginning of the line you want to select.
2. Press SHIFT + DOWN ARROW.  
The entire line, including the end-of-line character, is selected.
3. To extend your selection to include additional whole lines of text, repeat step 2.

### Deleting, Cutting, Copying, and Pasting Text

---

#### Deleting Text

Do one of the following to remove characters, selected text, or entire lines from your script.

- To remove a single character to the left of the insertion point, press BACKSPACE once; to remove a single character to the right of the insertion point, press DELETE once. To remove multiple characters, hold down BACKSPACE or DELETE.
- To remove text that you have selected, press BACKSPACE or DELETE.
- To remove an entire line, place the insertion point in that line and press CTRL+Y.

#### Cutting a Selection

To cut text from your script and place it on the Clipboard, press CTRL+X.

#### Copying a Selection

To copy text from your script and place it on the Clipboard, press CTRL+C.

## Pasting the Contents of the Clipboard into Your Script

To paste the contents of the Clipboard into your script:

1. Position the insertion point where you want to place the contents of the Clipboard.
2. Press CTRL+V.

## Adding Comments to a Script

---

There are two types of comments you can add to a script:

- Adding a Full-Line Comment
- Adding a Comment at the End of a Line of Code

### Adding a Full-Line Comment

Use the following procedure to designate an entire line as a comment.

1. Type an apostrophe ( ' ) at the start of the line.
2. Type your comment following the apostrophe.

When your script is run, the presence of the apostrophe at the start of the line will cause the entire line to be ignored.

### Adding a Comment at the End of a Line of Code

Use the following procedure to designate the last part of a line as a comment.

1. Position the insertion point in the empty space beyond the end of the line of code.
2. Type an apostrophe ( ' ).
3. Type your comment following the apostrophe.

When your script is run, the code on the first portion of the line will be executed, but the presence of the apostrophe at the start of the comment will cause the remainder of the line to be ignored.

## Finding and Replacing Text

---

### Finding Specified Text

Use the following procedure to locate instances of specified text quickly anywhere within your script.

1. Move the insertion point to where you want to start your search. (To start at the beginning of your script, press CTRL+HOME.)
2. Press CTRL+F.

The Script Editor displays the **Find** dialog box:



**Figure 10 Find Script Text Dialog Box**

3. In the **Find What** field, specify the text you want to find or select it from the list of previous searches.
4. Click **Find Next** or press ENTER.

The **Find** dialog box remains displayed, and the Script Editor either highlights the first instance of the specified text or indicates that it cannot be found.

5. If the specified text has been found, repeat step 4 to search for the next instance of it.

**Note:** If the **Find** dialog box blocks your view of an instance of the specified text, you can move the dialog box out of your way and continue with your search. You can also click **Cancel**, which removes the **Find** dialog box while maintaining the established search criteria, and then press F3 to find successive occurrences of the specified text. (If you press F3 when you have not previously specified text for which you want to search, the Script Editor displays the **Find** dialog box so you can specify the desired text.)

## Replacing Specified Text

Use the following procedure to automatically replace either all instances or selected instances of specified text.

1. Move the insertion point to where you want to start the replacement operation. (To start at the beginning of your script, press CTRL+HOME.)
2. Click **Edit > Replace**.

The Script Editor displays the **Replace** dialog box:



**Figure 11** *Replace Dialog Box*

3. In the **Find What** field, specify the text you want to replace or select it from the list of previous searches.
4. In the **Replace With** field, specify the replacement text or select it from the list of previous replacements.
5. To replace selected instances of the specified text, click **Find Next**. The Script Editor either highlights the first instance of the specified text or indicates that it cannot be found.
6. If the specified text has been found, either click **Replace** to replace that instance of it or click **Find Next** to highlight the next instance (if any).

Each time you click **Replace**, the Script Editor replaces that instance of the specified text and automatically highlights the next instance.

## Running, Pausing, and Stopping Your Script

---

### Running Your Script

To compile and run your script from within the Script Editor, click **Go** on the toolbar or press F5.

The script is compiled (if it has not already been compiled), the focus is switched to the parent window, and the script is executed.

**Note:** *During script execution, the Script Editor's application window is available only in a limited manner. Some of the menu commands may be disabled, and the toolbar tools may be inoperative.*

You can also use the Application Class ExecuteScript method to run scripts. See the *ExecuteScript* method for details.

### Pausing an Executing Script

To suspend the execution of a script that you are running, press CTRL+BREAK.

Execution of the script is suspended, and the instruction pointer (a gray highlight) appears on the line of code where the script stopped executing.

**Note:** *The instruction pointer designates the line of code that will be executed next if you resume running your script.*

### Stopping an Executing Script

Use the following procedure to stop the execution of a script that you are running.

1. If it is not paused, pause the script.
2. Click **StopDebugging** on the toolbar (or press SHIFT+F5).

**Note:** *Many of the functions of the Script Editor's application window may be unavailable while you are running a script. If you want to stop your script, but find that the toolbar is currently inoperative, press CTRL+BREAK to pause your script, then click **StopDebugging**.*

---

## Tracing Script Execution

---

### Stepping Through Your Script

Use the following procedure to trace the execution of your script with either the StepInto or StepOver method:

1. Do one of the following:
  - Click the StepInto or StepOver tool on the toolbar.
  - Press F11(StepInto) or F10 (StepOver).

The Script Editor places the instruction pointer on the sub main line of your script.

**Note:** *When you initiate execution of your script using either of these methods, the script will first be compiled, if necessary. Therefore, there may be a slight pause before execution actually begins. If your script contains any compile errors, it will not be executed. To debug your script, first correct any compile errors, and then execute it again.*

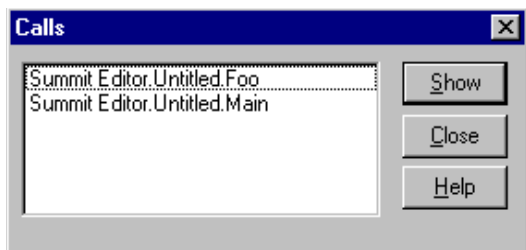
2. To continue tracing the execution of your script, repeat step 1.  
Each time you repeat step 1, the Script Editor executes the line or the procedure that contains the instruction pointer and then moves the instruction pointer to the next line or procedure to be executed.
3. When you finish tracing the execution of your script, either click **Go** on the toolbar (or press F5) to run the script at full speed or click **Stop Debugging** to halt execution of the script.

### Displaying the Calls Dialog Box

When you are stepping through a subroutine, you may need to determine the procedure calls by which you arrived at that point in your script. Use the following procedure to use the **Calls** dialog box to obtain this information.

1. Click **Calls** on the toolbar.

The Script Editor displays the **Calls** dialog box, which lists the procedure calls made by your script in the course of arriving at the present subroutine.



**Figure 12 Script Calls Dialog Box**

2. From the **Calls** dialog box, select the name of the procedure you wish to view.
3. Click the **Show** button.

The Script Editor highlights the currently executing line in the procedure you selected, scrolling that line into view if necessary. (During this process, the instruction pointer remains in its original location in the subroutine.)



## Setting and Removing Breakpoints

---

You set and remove breakpoints in your script as part of the debugging process.

### Starting Debugging Partway through a Script

Use the following procedure to begin the debugging process at a selected point in your script:

1. Place the insertion point in the line where you want to start debugging.
2. To set a breakpoint on that line, click **Toggle Breakpoint** on the toolbar (or press F9).

The line on which you set the breakpoint now appears in contrasting type.

3. Click **Go** on the toolbar (or press F5).

The Script Editor runs your script at full speed from the beginning and then pauses prior to executing the line containing the breakpoint. It places the instruction pointer on that line to designate it as the line that will be executed next when you either proceed with debugging or resume running the script.

### Continuing Debugging at a Line Outside the Current Subroutine

If you want to continue debugging at a line that *isn't* within the same subroutine, use the following procedure to move the instruction pointer to that line.

1. Place the insertion point in the line where you want to continue debugging.
2. To set a breakpoint on that line, press F9.
3. To run your script, click **Go** on the toolbar (or press F5).

The script executes at full speed until it reaches the line containing the breakpoint and then pauses with the instruction pointer on that line. You can now resume stepping through your script from that point.

### Debugging Selected Portions of Your Script

If you only need to debug parts of your script, use the following procedure to facilitate the task by using breakpoints.

1. Place a breakpoint at the start of each portion of your script that you want to debug.

**Note:** *Up to 255 lines in your script can contain breakpoints.*

2. To run the script, click **Go** on the toolbar or press F5.

The script executes at full speed until it reaches the line containing the first breakpoint and then pauses with the instruction pointer on that line.

3. Step through as much of the code as you need to.

4. To resume running your script, click **Go** on the toolbar or press F5.

The script executes at full speed until it reaches the line containing the second breakpoint and then pauses with the instruction pointer on that line.

5. Repeat steps 3 and 4 until you have finished debugging the selected portions of your script.

### Removing a Single Breakpoint Manually

Use the following procedure to delete breakpoints manually one at a time.

1. Place the insertion point on the line containing the breakpoint that you want to remove.

2. Click **Toggle Breakpoint** on the toolbar, or press F9.

The breakpoint is removed, and the line no longer appears in contrasting type.

### Removing All Breakpoints Manually

To delete all breakpoints manually in a single operation, click **Debugger > Clear All Breakpoints**.

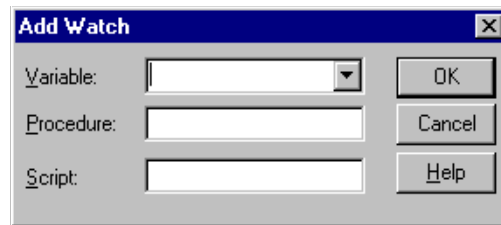
## Working with Watch Variables

Watch variables allow you to track the changing values of variables in a script.

### Adding Watch Variables

Use the following procedure to add a variable to the Script Editor's watch variable list.

1. Click **Add Watch** on the toolbar or press SHIFT+F9.  
The Script Editor displays the **Add Watch** dialog box.



**Figure 13 Add Watch Dialog Box**

2. Use the controls in the Context box to specify where the variable is defined (locally, publicly, or privately) and, if it is defined locally, in which routine it is defined.
3. In the **Variable Name** field, enter the name of the variable you want to add to the watch variable list.

You can only watch variables of fundamental data types, such as Integer, Long, Variant, and so on; you cannot watch complex variables such as structures or arrays. You can, however, watch individual elements of arrays or structure members.

Use the following syntax to watch individual elements of arrays or structure members in a script:

```
[variable [(index,...)] [.member [(index,...)]]...
```

Where **variable** is the name of the structure or array variable, **index** is a literal number, and **member** is the name of a structure member.

For example, the following are valid watch expressions:

**Table 6 Sample Watch Expressions**

Watch Variable	Description
<code>a(1)</code>	Element 1 of array <b>a</b>
<code>person.age</code>	Member <b>age</b> of structure <b>person</b>
<code>company(10,23).person.age</code>	Member <b>age</b> of structure <b>person</b> that is at element 10,23 within the array of structures called <b>company</b>

**Note:** *If you are executing the script, you can display the names of all the variables that are “in scope,” or defined within the current function or subroutine, on the drop-down Variable Name list and select the variable you want from that list.*

4. Click **OK** or press ENTER.

If this is the first variable you are placing on the watch variable list, the watch pane opens far enough to display that variable. If the watch pane was already open, it expands far enough to display the variable you just added.

**Note:** *Although you can add as many watch variables to the list as you want, the watch pane only expands until it fills half of the Script Editor’s application window. If your list of watch variables becomes longer than that, you can use the watch pane’s scroll bars to bring hidden portions of the list into view.*

### Selecting Variables on the Watch List

In order to delete a variable from the Script Editor's watch variable list or modify the value of a variable on the list, do one of the following:

- Place the mouse pointer on the variable you want to select and click the left mouse button.
- If one of the variables on the watch list is already selected, use the arrow keys to move the selection highlight to the desired variable.

- If the insertion point is in the edit pane, press F6 to highlight the most recently selected variable on the watch list and then use the arrow keys to move the selection highlight to the desired variable.

**Note:** Pressing F6 again returns the insertion point to its previous position in the edit pane.

## Deleting Watch Variables

Use the following procedure to delete a selected variable from the Script Editor's watch variable list.

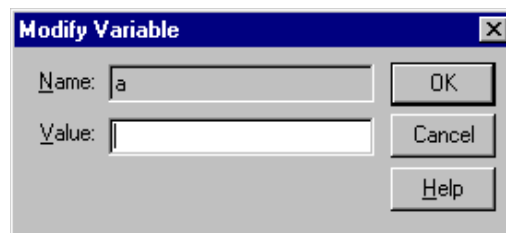
1. Select the variable on the watch list.
2. Click **Debugger > Delete Watch**, or press DELETE.

## Modifying the Value of Variables on the Watch Variable List

When the debugger has control, you can modify the value of any of the variables on the Script Editor's watch variable list. Use the following procedure to change the value of a selected watch variable.

1. Place the mouse pointer on the name of the variable whose value you want to modify and double-click the left mouse button.
2. Select the name of the variable whose value you want to modify and press ENTER or F2.

The Script Editor displays the **Modify Variable** dialog box.



**Figure 14** *Modify Variable Dialog Box*

**Note:** The name of the variable you selected on the watch variable list appears in the *Name* field.

When you use the **Modify Variable** dialog box to change the value of a variable, you don't have to specify the context. The Script Editor first searches locally for the definition of that variable, then privately, then publicly.

3. Enter the new value for your variable in the **Value** field.
4. Click **OK**.  
The new value of your variable appears on the watch variable list.

### Compiling Your Script

---

Use the following procedure to create compiled script files from your script source:

1. Click **Tools > Open Script** and select the file that contains the script you want to compile.
2. Click **Debugger > Compile**, or press F7.
3. Enter the name of the file in which to save the compiled script and select **OK**.

The script is compiled and saved in a file with a .ebx extension.

**Note:** You can also use the *Application.CompileScriptFile* method to compile scripts. Check the *Extensibility Reference* or the *Extensibility Online Help* for more details.

### Using Interscript Calls

---

#### Guidelines for Using a Script to Call Another Script

You can write a script that includes code that calls and executes another script. The following guidelines apply to this process:

- You can only call and execute a compiled script from within another script.
- Use the `LoadScript` method to load the script into memory.
- Use the `FreeScript` to unload the script from memory.
- Even if you call `LoadScript` multiple times, the script is only loaded into memory one time. However, for each `LoadScript` call you make, you must include a corresponding `FreeScript` call. If you do not do this, the script will not be unloaded from memory.

## Debugging Interscript Calls

Use the following procedure to debug a script that uses interscript calls:

1. Enter the call to the compiled script you are including and set a breakpoint on the call.
2. Click **Debugger > StepInto**.

The Script Editor displays the source code for the compiled script you are calling, and steps through it line by line.

When the trace of the called script is complete, the Script Editor redisplay the calling script.

**Note:** *The script you are calling must be compiled with debugging turned on. See [Compiling Your Script](#), earlier in this chapter, for details.*

## Working with the Dialog Editor

---

### Inserting a Dialog Box into Your Script

To insert a dialog box into your script:

1. Place the insertion point where you want the BasicScript code for the dialog box to appear in your script.
2. Click **Edit > Insert Dialog**.

The Script Editor's application window is temporarily disabled, and Dialog Editor appears, displaying a new dialog box in its application window.

3. Use the Dialog Editor to create your dialog box.
4. Exit and Return from Dialog Editor and return to the Script Editor. The Script Editor automatically places the code for the dialog box in your script at the location of the insertion point.

### Editing an Existing Dialog Box

To edit an existing dialog box template in your script:

1. Select the BasicScript code for the entire dialog box template.
2. Click **Edit > Edit Dialog**.

The Script Editor's application window is temporarily disabled, and Dialog Editor appears, displaying in its application window a dialog box created from the code you selected.

3. Use the Dialog Editor to modify your dialog box.
4. Exit from the Dialog Editor and return to the Script Editor.

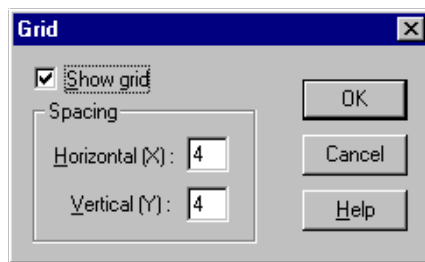
The Script Editor automatically replaces the BasicScript code you originally selected with the revised code generated by the Dialog Editor.

### Displaying and Adjusting the Grid

Use the following procedure to display and adjust the X and Y settings, which can help you position controls more precisely within your dialog box:

1. Press CTRL+G.

The Dialog Editor displays the following dialog box:



**Figure 15 Grid Dialog Box**

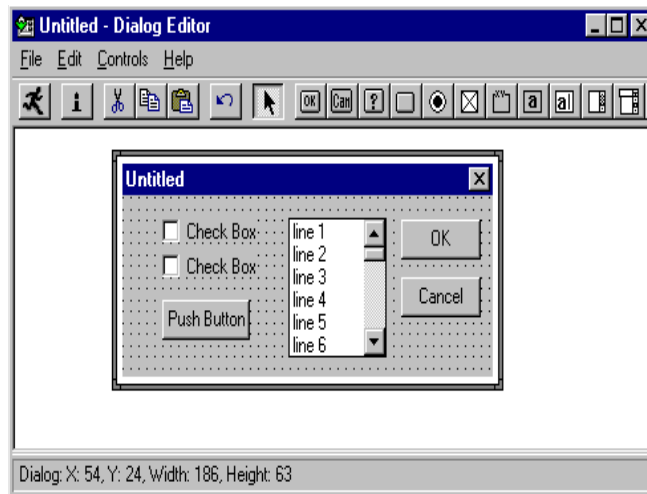
2. To display the grid in your dialog box, click **Show grid**.
3. To change the current X and Y settings, enter new values in the X and Y fields.

**Note:** The values of X and Y in the **Grid** dialog box determine the grid's spacing. Assigning smaller X and Y values produces a more closely spaced grid, which enables you to move the mouse pointer in smaller horizontal and vertical increments as you position controls. Assigning larger X and Y values produces the opposite effect on both the grid's spacing and the movement of the mouse pointer. The X and Y settings entered in the **Grid** dialog box remain in effect regardless of whether you choose to display the grid.

4. Click **OK** or press ENTER.



The Dialog Editor displays the grid with the settings you specified.



**Figure 16** Dialog Editor with Grid Displayed

5. With the grid displayed, line up the crosshairs on the mouse pointer with the dots on the grid to position controls precisely and align them with respect to other controls.

## Changing Titles and Labels

Use the following procedure to change the title of a dialog box, as well as the labels of group boxes, option buttons, push buttons, text controls, and check boxes:

1. Display the **Information** dialog box for the dialog box whose title you want to change or for the control whose label you want to change.
2. Enter the new title or label in the **Text\$** field.

**Note:** Dialog box titles and control labels are optional. Therefore, you can leave the **Text\$** field blank.

3. If the information in the **Text\$** field should be interpreted as a variable name rather than a literal string, click **Variable Name**.
4. Click **OK** or press ENTER.

The new title or label is now displayed on the title bar or on the control.

## Assigning Accelerator Keys

Use the following procedure to designate a letter from a control's label to serve as the accelerator key for that control.

1. Display the **Information** dialog box for the control to which you want to assign an accelerator key.
2. In the **Text\$** field, type an ampersand (&) before the letter you want to designate as the accelerator key.
3. Click **OK** or press ENTER.

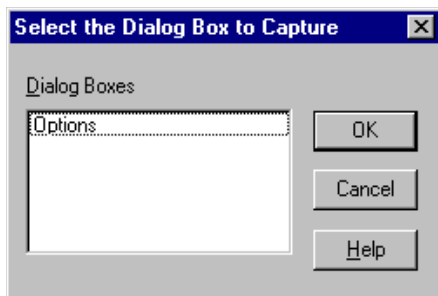
The letter you designated is now underlined on the control's label, and users will be able to access the control by pressing ALT + the underlined letter.

## Capturing Standard Windows Dialog Boxes

Use the following procedure to capture the standard Windows controls from any standard Windows dialog box in another application, and insert those controls into the Dialog Editor for editing:

1. Display the dialog box you want to capture.
2. Open the Dialog Editor.
3. Click **File > Capture Dialog**.

The Dialog Editor displays a dialog box that lists all open dialog boxes that it is able to capture:



**Figure 17** Capturing a Dialog Box

4. Select the dialog box that you want to capture, then click **OK**.

**Note:** The Dialog Editor only supports standard Windows controls and standard Windows dialog boxes. Therefore, if the target dialog box contains both standard Windows controls and custom controls,

*only the standard Windows controls will appear in the Dialog Editor's application window. If the target dialog box is not a standard Windows dialog box, you will be unable to capture the dialog box or any of its controls.*

## Testing Your Dialog Boxes

The Dialog Editor lets you run your edited dialog box purposes. When you click **Test**, your dialog box comes alive, which gives you an opportunity to make sure it functions properly and fix any problems before you incorporate the dialog box template into your script.

Before you run your dialog box, take a moment to look it over for basic problems such as the following:

- Does the dialog box contain a command button—that is, a default OK or Cancel button, a push button, or a picture button?
- Does the dialog box contain all the necessary push buttons?
- Does the dialog box contain a Help button if one is needed?
- Are the controls aligned and sized properly?
- If there is a text control, is its font set properly?
- Are the close box and title bar displayed (or hidden) as you intended?
- Are the control labels and dialog box title spelled and capitalized correctly?
- Do all the controls fit within the borders of the dialog box?
- Could you improve the design of the dialog box by adding one or more group boxes to set off groups of related controls?
- Could you clarify the purpose of any unlabeled control (such as a text box, list box, combo box, drop list box, picture, or picture button) by adding a text control to serve as a de facto label for it?
- Have you made all the necessary accelerator key assignments?
- After you've fixed any elementary problems, you're ready to run your dialog box so you can check for problems that don't become apparent until a dialog box is activated.

Testing your dialog box is an iterative process that involves running the dialog box to see how well it works, identifying problems, stopping the test and fixing those problems, then running the dialog box again to

make sure the problems are fixed and to identify any additional problems, and so forth—until the dialog box functions the way you intend.

Use the following procedure to test your dialog box and fine-tune its performance:

1. Click **Run** on the toolbar, or press F5, to make the dialog box operational.
2. Check the dialog box's functions.
3. To stop the test, click **Run**, press F5, or double-click the dialog box's close box (if it has one).
4. Make any necessary adjustments to the dialog box.
5. Repeat steps 1-4 as many times as you need in order to get the dialog box working properly.

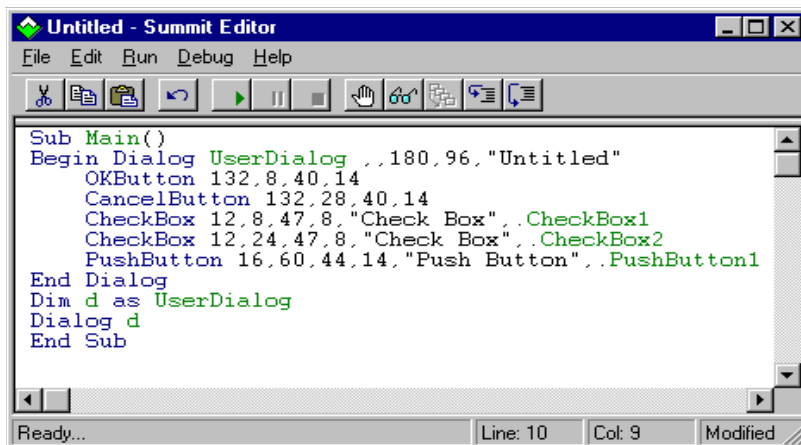
### Incorporating Dialog Boxes or Controls into Your Script

You create dialog boxes and dialog box controls in the Dialog Editor. To incorporate them into a script, you copy them to the Clipboard. When you copy the dialog to the Clipboard, it is stored in the form of Basic Script statements. You then paste the contents of the Clipboard into the script.

Use the following procedure to incorporate a dialog box or control into your script:

1. Select the dialog box or control that you want to incorporate into your script.
2. Press CTRL+C.
3. Open your script and paste in the contents of the Clipboard at the desired point.

The dialog box template or control is now described in BasicScript statements in your script, as shown in the following example:



```

Sub Main()
Begin Dialog UserDialog , ,180,96,"Untitled"
  OKButton 132,8,40,14
  CancelButton 132,28,40,14
  CheckBox 12,8,47,8,"Check Box",.CheckBox1
  CheckBox 12,24,47,8,"Check Box",.CheckBox2
  PushButton 16,60,44,14,"Push Button",.PushButton1
End Dialog
Dim d as UserDialog
Dialog d
End Sub

```

**Figure 18 Sample Dialog Box in Basic Script**

## Selecting Controls

Do one of the following to select a control in a dialog box:

- With the **Pick** tool active, place the mouse pointer on the desired control and click the mouse button.
- With the **Pick** tool active, press the TAB key repeatedly until the focus moves to the desired control.

The control is now surrounded by a thick frame to indicate that it is selected and you can edit it.

## Selecting Dialog Boxes

Do one of the following to select an entire dialog box:

- With the **Pick** tool active, place the mouse pointer on the title bar of the dialog box or on an empty area within the borders of the dialog box (that is, on an area where there are no controls) and click the mouse button.
- With the **Pick** tool active, press the TAB key repeatedly until the focus moves to the dialog box.

The dialog box is now surrounded by a thick frame to indicate that it is selected and you can edit it.

### Repositioning Items

#### Repositioning Items with the Mouse

Use the following procedure to reposition items in a dialog box or control by dragging it with the mouse:

1. With the **Pick** tool active, place the mouse pointer on an empty area of the dialog box or on a control.
2. Depress the mouse button and drag the dialog box or control to the desired location.

**Note:** *The increments by which you can move a control with the mouse are governed by the grid setting. For example, if the grid's X setting is 4 and its Y setting is 6, you'll be able to move the control horizontally only in increments of 4 X units and vertically only in increments of 6 Y units. This feature is handy if you're trying to align controls in your dialog box. If you want to move controls in smaller or larger increments, press CTRL+G to display the **Grid** dialog box and adjust the X and Y settings.*

#### Repositioning Items with the Arrow Keys

Use the following procedure to reposition items in a dialog box or control by dragging it with the arrow keys:

1. Select the dialog box or control that you want to move.
2. Do one of the following:
  - Press an arrow key once to move the item by 1 X or Y unit in the desired direction.
  - Steadily press an arrow key to “nudge” the item gradually along in the desired direction.

**Note:** *When you reposition an item with the arrow keys, a faint, partial afterimage of the item may remain visible in the item's original position. These afterimages are rare and will disappear once you test your dialog box.*

## Repositioning Dialog Boxes with the Dialog Information Dialog Box

Use the following procedure to reposition items in a dialog box or control by using the **Dialog Information** dialog box.

1. Display the **Dialog Box Information** dialog box.

**Note:** For information on displaying the **Dialog Information** dialog box, see *Displaying the Dialog Information Dialog Box*, later in this chapter.

2. Do one of the following:
  - Change the X and Y coordinates in the **Position** group box.
  - Leave the X and/or Y coordinates blank.
3. Click **OK** or press ENTER.

If you specified X and Y coordinates, the dialog box moves to that position. If you left the X coordinate blank, the dialog box will be centered horizontally relative to the parent window of the dialog box when the dialog box is run. If you left the Y coordinate blank, the dialog box will be centered vertically relative to the parent window of the dialog box when the dialog box is run.

## Repositioning Controls with the Dialog Information Dialog Box

1. Use the following procedure to move a selected control by changing its coordinates in the **Dialog Information** dialog box for that control.

**Note:** For information on displaying the **Dialog Information** dialog box, see *Displaying the Dialog Information Dialog Box*, later in this chapter.

2. Display the **Information** dialog box for the control that you want to move.
3. Change the X and Y coordinates in the **Position** group box.
4. Click **OK** or press ENTER.

The control moves to the specified position.

## Resizing Items

### Resizing Items with the Mouse

Use the following procedure to change the size of a selected dialog box or control by dragging its borders or corners with the mouse:

1. With the **Pick** tool active, select the dialog box or control that you want to resize.
2. Place the mouse pointer over a border or corner of the item.
3. Depress the mouse button and drag the border or corner until the item reaches the desired size.

### Resizing Items with the Information Dialog Box

Use the following procedure to change the size of a selected dialog box or control by changing its **Width** or **Height** settings in the **Information** dialog box.

1. Display the **Information** dialog box for the dialog box or control that you want to resize.
2. Change the **Width** and **Height** settings in the **Size** group box.
3. Click the **OK** button or press ENTER.

The dialog box or control is resized to the dimensions you specified.

### Resizing Selected Items Automatically

Use the following procedure to adjust the borders of certain controls automatically to fit the text displayed on them.

To resize selected controls automatically:

1. With the **Pick** tool active, select the option button, text control, push button, check box, or text box that you want to resize.
2. Press F2.

The borders of the control will expand or contract to fit the text displayed on it.



## Adding Controls

Use the following procedure to add one or more controls to your dialog box using simple mouse and keyboard methods.

1. From the toolbar, choose the tool corresponding to the type of control you want to add.

**Note:** When you pass the mouse pointer over an area of the display where a control can be placed, the pointer becomes an image of the selected control with crosshairs (for positioning purposes) to its upper left. The name and position of the selected control appear on the status bar. When you pass the pointer over an area of the display where a control cannot be placed, the pointer changes into a circle with a slash through it (the “prohibited” symbol).

**Note:** You can only insert a control within the borders of the dialog box you are creating. You cannot insert a control on the dialog box’s title bar or outside its borders.

2. Place the pointer where you want the control to be positioned and click the mouse button.

The control you just created appears at the specified location. (To be more specific, the upper left corner of the control will correspond to the position of the pointer’s crosshairs at the moment you clicked the mouse button.) The control is surrounded by a thick frame, which means that it is selected, and it may also have a default label.

After the new control has appeared, the mouse pointer becomes an arrow, to indicate that the Pick tool is active and you can once again select any of the controls in your dialog box.

3. To add another control of the same type as the one you just added, press CTRL+D.

A duplicate copy of the control appears.

4. To add a different type of control, repeat steps 1 and 2.

5. To reactivate the Pick tool, do one of the following:

- Click the arrow-shaped tool on the toolbar.
- Place the mouse pointer on the title bar of the dialog box or outside the borders of the dialog box (that is, on any area where the mouse pointer turns into the “prohibited” symbol) and click the mouse button.

### Duplicating Controls

Use the following procedure to use the Dialog Editor's duplicating feature, which saves you the work of creating additional controls individually if you need one or more copies of a particular control:

1. Select the control that you want to duplicate.
2. Press CTRL+D.

A duplicate copy of the selected control appears in your dialog box.

3. Repeat step 2 as many times as necessary to create the desired number of duplicate controls.

### Adding Pictures to a Dialog

You can add pictures to a dialog from a file or from a picture library.

#### Adding Pictures from Files

Use the following procedure to display a Windows bitmap or metafile from a file on a picture control or picture button control by using the control's **Information** dialog box to indicate the file in which the picture is contained.

1. Display the **Information** dialog box for the picture control or picture button control whose picture you want to specify.
2. In the **Picture source** option button group, click **File**.
3. In the **Name\$** field, enter the name of the file containing the picture you want to display in the picture control or picture button control.

**Note:** *By clicking the **Browse** button, you can display the **Select a Picture File** dialog box and use it to find the file.*

4. Click the **OK** button or press ENTER.

The picture control or picture button control now displays the picture you specified.

## Adding Pictures from Picture Libraries

Use the following procedure to display a Windows bitmap or metafile from a file on a picture control or picture button control by using the control's **Information** dialog box to indicate the file in which the picture is contained.

1. Display the **Information** dialog box for the picture control or picture button control whose picture you want to specify.
2. In the **Picture source** option button group, click **File**.
3. In the **Name\$** field, enter the name of the file containing the picture you want to display in the picture control or picture button control.

**Note:** *By clicking the **Browse** button, you can display the **Select a Picture File** dialog box and use it to find the file.*

4. Click **OK** or press ENTER.  
The picture control or picture button control now displays the picture you specified.

## Pasting Items into Dialog Editor

### Pasting Existing Dialog Boxes into Dialog Editor

If you want to modify a **BasicScript** dialog box template contained in your script, use the following procedure to select the template and paste it into dialog editor for editing:

1. Copy the entire **BasicScript** dialog box template (from the `Begin Dialog` instruction to the `End Dialog` instruction) from your script to the Clipboard.
2. Open the Dialog Editor.
3. Press CTRL+V.
4. When the Dialog Editor asks whether you want to replace the existing dialog box, click **Yes**.

The Dialog Editor creates a new dialog box corresponding to the template contained on the Clipboard.

### Pasting Controls from Existing Dialog Boxes into Dialog Editor

If you want to modify the BasicScript statements in your script that correspond to one or more dialog box controls, use the following procedure to select the statements and paste them into Dialog Editor for editing:

1. Copy the BasicScript description of the control(s) from your script to the Clipboard.
2. Open Dialog Editor.
3. Press CTRL+V.

Dialog Editor adds to your current dialog box one or more controls corresponding to the description contained on the Clipboard.

### Displaying the Information Dialogs

There are two types of **Information** dialog boxes:

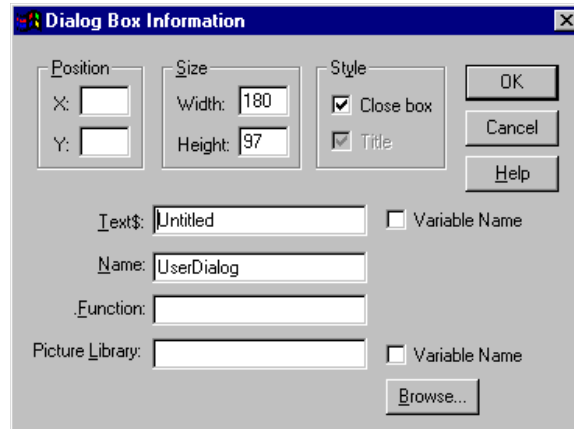
- Information Dialog Box for Dialogs
- Information Dialog Box for Controls

#### Displaying the Information Dialog Boxes for Dialogs

Do one of the following to display the **Dialog Box Information** dialog box to check and adjust attributes that pertain to the dialog box as a whole:

- With the Pick tool active, place the mouse pointer on an area of the dialog box where there are no controls and double-click the mouse button.
- With the Pick tool active, select the dialog box and either click the **Information** tool on the toolbar, press ENTER, or press CTRL+I.

The following figure shows the **Dialog Box Information** dialog box:



**Figure 19** *Dialog Box Information Dialog Box*

## Attributes You Can Adjust with the Dialog Box Information Dialog Box

The **Dialog Box Information** dialog box can be used to check and adjust the following attributes, which pertain to the dialog box as a whole.

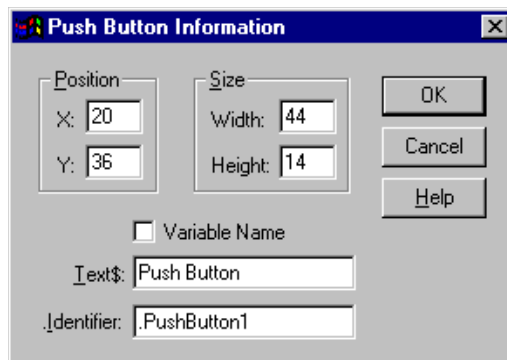
- **Position** (optional): X and Y coordinates on the display, in dialog units
- **Size** (mandatory): width and height of the dialog box, in dialog units
- **Style** (optional): options that allow you to determine whether the close box and title bar are displayed
- **Text\$** (optional): text displayed on the title bar of the dialog box
- **Name** (mandatory): name by which you refer to this dialog box template in your BasicScript code
- **Function** (optional): name of a BasicScript function in your dialog box
- **Picture Library** (optional): picture library from which one or more pictures in the dialog box are obtained

## Displaying the Information Dialog Boxes for Controls

Do one of the following to display the **Information** dialog box for a control to check and adjust attributes that pertain to that particular control.

- With the **Pick** tool active, place the mouse pointer on the desired control and double-click the mouse button.
- With the **Pick** tool active, select the control and either click the **Information** tool on the toolbar, press ENTER, or press CTRL+I.

The Dialog Editor displays an **Information** dialog box corresponding to the control you selected. For example:



**Figure 20 Control Information Dialog Box**

## Attributes You Can Adjust with the Information Dialog Boxes for Controls

**Control Information** dialog boxes can be used to check and adjust the attributes of the following controls:

- **Default OK Button Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code

- **Default Cancel Button Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
- **Help Button Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **FileName\$** (optional): Name of the help file that you want to invoke
  - **Context&** (mandatory): The context ID specifying which help topic to jump to
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
- **Push Button Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Text\$** (optional): text displayed on a control
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
- **Option Button Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Text\$** (optional): text displayed on a control
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
  - **.Option Group** (mandatory): name by which you refer to a group of option buttons in your BasicScript code

- **Check Box Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Text\$** (optional): text displayed on a control
  - **.Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog box has been processed
- **Group Box Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Text\$** (optional): text displayed on a control
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
- **Text Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Text\$** (optional): text displayed on a control
  - **Font** (optional): font in which text is displayed
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
- **Text Box Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Multiline** (optional): option that allows you to determine whether users can enter a single line of text or multiple lines
  - **.Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog box has been processed



- **List Box Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog box has been processed
  - **Array\$** (mandatory): name of an array variable in your BasicScript code
- **Combo Box Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog box has been processed
  - **Array\$** (mandatory): name of an array variable in your BasicScript code
- **Drop List Box Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog box has been processed
  - **Array\$** (mandatory): name of an array variable in your BasicScript code
- **Picture Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code

- **.Identifier** (optional): name of the file containing a picture that you want to display or the name of a picture that you want to display from a specified picture library
- **Frame** (optional): option that allows you to display a 3-D frame
- **Picture Button Information** dialog box
  - **Position** (mandatory): X and Y coordinates within the dialog box, in dialog units
  - **Size** (mandatory): width and height of the control, in dialog units
  - **.Identifier** (optional): name by which you refer to a control in your BasicScript code
  - **.Identifier** (optional): name of the file containing a picture that you want to display or the name of a picture that you want to display from a specified picture library



---

## Appendix A

# Rational Rose Script Editor Shortcuts

---

This appendix identifies the shortcuts that can be used with the Rational Script Editor.

## General Shortcuts

---

**Table 7** *General Shortcuts*

---

<b>Key Name(s)</b>	<b>Description</b>
F1	Provides context-sensitive help for selected menu commands and variables in the watch pane, for BasicScript terms in the edit pane that have been selected or that contain the insertion point, and for displayed dialog boxes.
CTRL+F	Displays the <b>Find</b> dialog box, which allows you to specify text for which you want to search.
F3	Searches for the next occurrence of previously specified text. If you have not previously specified text for which you want to search, displays the <b>Find</b> dialog box.
ESC	Deactivates the <b>Help</b> pointer if it is active. Otherwise, compiles your script and returns you to the host application.

---

## Navigating Shortcuts

---

**Table 8 Navigating Shortcuts**

<b>Key Name(s)</b>	<b>Description</b>
UP ARROW	Moves the insertion point up one line.
DOWN ARROW	Moves the insertion point down one line.
LEFT ARROW	Moves the insertion point left by one character position.
RIGHT ARROW	Moves the insertion point right by one character position.
PAGE UP	Moves the insertion point up by one window.
PAGE DOWN	Moves the insertion point down by one window.
CTRL+PAGE UP	Scrolls the insertion point left by one window.
CTRL+PAGE DOWN	Scrolls the insertion point right by one window.
CTRL+LEFT ARROW	Moves the insertion point to the start of the next word to the left.
CTRL + RIGHT ARROW	Moves the insertion point to the start of the next word to the right.
HOME	Places the insertion point before the first character in the line.
END	Places the insertion point after the last character in the line.
CTRL+HOME	Places the insertion point before the first character in the script.
CTRL+END	Places the insertion point after the last character in the script.

---

---

## Editing Shortcuts

---

**Table 9** *Editing Shortcuts*

<b>Key Name(s)</b>	<b>Description</b>
DELETE	Removes the selected text or removes the character following the insertion point without placing it on the Clipboard.
BACKSPACE	Removes the selected text or removes the character preceding the insertion point without placing it on the Clipboard.
CTRL+Y	Deletes the entire line containing the insertion point without placing it on the Clipboard.
TAB	Inserts a tab character.
ENTER	Inserts a new line, breaking the current line.
CTRL+C	Copies the selected text, without removing it from the script, and places it on the Clipboard.
CTRL+X	Removes the selected text from the script and places it on the Clipboard.
CTRL+V	Inserts the contents of the Clipboard at the location of the insertion point.
SHIFT + any navigating shortcut	Selects the text between the initial location of the insertion point and the point to which the keyboard shortcut would normally move the insertion point. (For example, pressing SHIFT + CTRL + LEFT ARROW selects the word to the left of the insertion point; pressing SHIFT+CTRL+HOME selects all the text from the location of the insertion point to the start of your script.)
CTRL+Z	Reverses the effect of the preceding editing change(s).

## Debugging Shortcuts

---

**Table 10** *Debugging Shortcuts*

<b>Key Name(s)</b>	<b>Description</b>
SHIFT+F9	Displays the <b>Add Watch</b> dialog box, in which you can specify the name of a BasicScript variable. The Script Editor then displays the value of that variable, if any, in the watch pane of its application window.
ENTER or F2	Displays the <b>Modify Variable</b> dialog box for the selected watch variable, which enables you to modify the value of that variable.
F6	If the watch pane is open, switches the insertion point between the watch pane and the edit pane.
CTRL+BREAK	Suspends execution of an executing script and places the instruction pointer on the next line to be executed.
F9	Sets or removes a breakpoint on the line containing the insertion point.
F10	Activates the Step Over command, which executes the next line of a BasicScript script and then suspends execution of the script. If the script calls another BasicScript procedure, BasicScript will run the called procedure in its entirety.
F11	Activates the Step Into command, which executes the next line of a BasicScript script and then suspends execution of the script. If the script calls another BasicScript procedure, execution will continue into each line of the called procedure.

---

## File Menu Shortcuts

---

**Table 11** *File Menu Shortcuts*

<b>Key Name(s)</b>	<b>Description</b>
CTRL+W	Compiles your script and returns you to the host application.
CTRL+S	Saves the currently open script.

---

---

## Edit Menu Shortcuts

---

**Table 12** *Edit Menu Shortcuts*

<b>Key Name(s)</b>	<b>Description</b>
CTRL+Z	Reverses the effect of the preceding editing change(s).
CTRL+X	Removes the selected text from the script and places it on the Clipboard.
CTRL+C	Copies the selected text, without removing it from the script, and places it on the Clipboard.
CTRL+V	Inserts the contents of the Clipboard at the current position of the insertion point.
CTRL+A	Selects all the text in the edit window.
CTRL+F	Displays the <b>Find</b> dialog box, which allows you to specify text for which you want to search. Remembers and allows you to choose from a list of previous search strings.
CTRL+H	Displays the <b>Replace</b> dialog box, which allows you to substitute replacement text for instances of specified text. Remembers and allows you to choose from a list of previous search and replace strings.
CTRL+G	Presents the <b>Goto Line</b> dialog box, which allows you to move the insertion point to the start of a specified line number in your script.

---

## Debugger Menu Shortcuts

---

**Table 13** *Debugger Menu Shortcuts*

<b>Key Name(s)</b>	<b>Description</b>
F5	Runs the current script.
CTRL+SHIFT+F5	Restarts the current script beginning with the line at which it was stopped using the Break command.
SHIFT+F5	Stops script execution.
F11	Steps through the script code line by line, tracing into called procedures.
F10	Steps through the script code line by line without tracing into called procedures.
F7	Compiles the current script without executing it.
SHIFT+F9	Displays the <b>Add Watch</b> dialog box, in which you can specify the name of a BasicScript variable. That variable, together with its value (if any), is then displayed in the watch pane of the Script Editor's application window.
DELETE	Deletes a selected variable from the watch variable list.
ENTER	Displays the <b>Modify Variable</b> dialog box for a selected variable, which enables you to modify the value of that variable.
F9	Toggles a breakpoint on the line containing the insertion point.

---





## *Appendix B*

# *Developing Add-Ins for Rational Rose*

---

## **Introduction**

---

This appendix is provided to give additional information for customers wanting to explore the use of add-ins. However, creation of add-ins is not directly supported by Rational Technical Support. Additional support for add-ins is available through the Rational Unified Solutions Partner Program and Rational University.

For more information on the Rational Unified Solutions Partner Program see:

<http://www.rational.com/corpinfo/partners/>

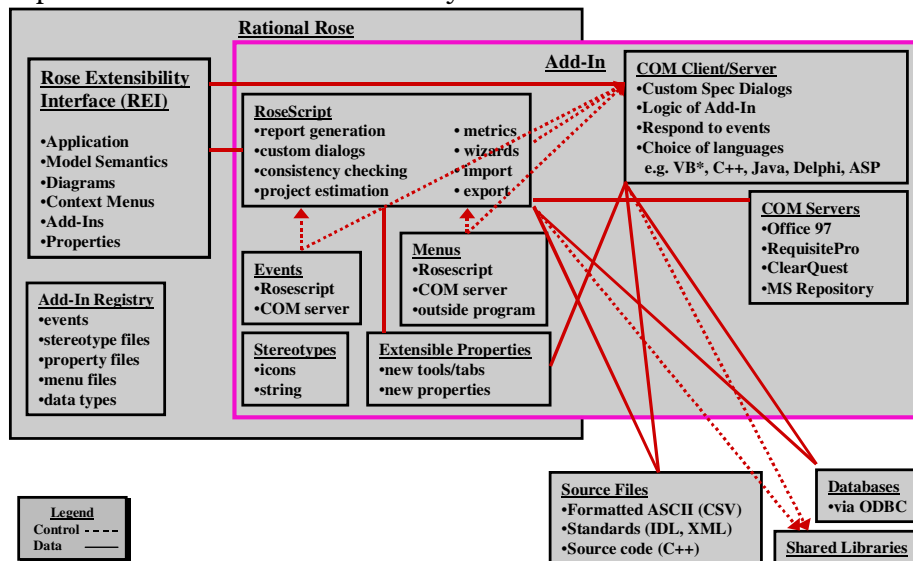
For training on Rational Rose's REI and add-ins see the "Extending Rational Rose" course from Rational University:

<http://www.rational.com/university/description/>

Add-in's allow you to package customizations and automation of several Rational Rose features through the Extensibility Interface (REI) into one package. An add-in is a collection of some combination of the following:

- Main menu items
- Shortcut menu items
- Custom specifications
- Properties
- Data types
- Stereotypes
- Online help
- Context-sensitive help
- Event handling
- Functionality through Rational Rose Scripts or controls (OLE-server)

Rational Rose Script or any language such as VB or C++ that can expose itself as an OLE server may be used to build an Add-in.



**Figure 21 Rational Rose Add-Ins Architecture**

**For more information, see page 101.**

**Note:** Servers that wish to use Rational Rose must use the supplied *typelib* included with Rational Rose.

## Why Create Add-Ins?

---

You might want to create an add-in as opposed to a script or program if you answer “yes” to any of the following questions:

- Do you want to take advantage of Rational Rose events like `OnNewModel`, `OnAppInit`?
- Do you want to interact with other Rational Rose add-ins?

## Types of Add-Ins

---

There are two types of add-ins: basic and language. They are defined below:

- **Basic:** A basic add-in is a non-language add-in that supplies its own responses for events to execute third-party scripts or executables, such as a Visual Basic program. It does not use the component view for code generation. A basic add-in cannot register for certain code generation-related events.
- **Language:** A language add-in takes advantage of the mapping to components by defining a target language. It also supplies its own responses for events that pertain to code generation and round-trip engineering integration. Code generation and round-trip engineering events include `OnGenerateCode`, `OnBrowseBody`, and `OnBrowseHeader`. Language add-ins support custom data types and overriding the default specification.

## What is in an add-in?

---

Add-in's customize or contain one or more of the following:

- Main menus
- Shortcut menu
- Custom specifications
- Properties

---

**For more information, see page 101.**

- Data types
- Stereotypes
- Online help
- Context-sensitive help
- Registering for events
- Functionality

Each of these are explained in the next sections.

### Main menus

The Rational Rose main menus are the menus at the top of the Rational Rose window, such as File, Edit, etc. These menus connect the user interface to functionality in Rational Rose. You can customize these menus to link functionality in your add-in to the Rational Rose user interface.

### Shortcut menu

The Rational Rose shortcut menu displays whenever you or your user right-clicks on part of the user interface. The shortcut menu is another link between the user interface and functionality in Rational Rose. You can customize this menu to link functionality in your add-in to the Rational Rose user interface.

### Custom Specifications

Rational Rose displays a standard specification dialog for each model element to allow definition and description of that model element. If you are writing a language add-in, you can override the standard Rational Rose specification dialog and display your own custom dialog. This is useful to:

- Remove non-relevant or inappropriate information
- Target the dialog to your end-user's needs
- Drive the dialog by stereotype or other characteristics, for example, naming conventions

---

**For more information, see page 101.**

## Properties

Rational Rose model properties allow you to extend Rational Rose model elements through additional properties and their values. You can add custom tools (a tab on the specification dialog), sets, and properties to store the information relevant to your add-in with each Rational Rose model element. You can also use this information to determine when functionality in your add-in should occur.

## Data types

Rational Rose data types allow you to customize which selections your user sees in the type drop down boxes for model elements associated with your add-in.

## Stereotypes

Rational Rose stereotypes allow you to customize the look of different model elements as makes sense to your add-in. This custom look can be as simple as an additional text string (for example, <<Special Class>>), or as fancy as new icons for the diagram editor, toolbar buttons, and browser icons.

## Online help

Rational Rose provides extensive online help to explain the product. You can also add your online help.

## Context-sensitive help

Rational Rose provides context-sensitive help to provide quick, brief information on the context. You can add context-sensitive help to your add-in's user interface. and your custom Rational Rose main menu items, shortcut menu items, and properties.

## Registering for events

Rational Rose provides several COM events for which your add-in can register and respond.

---

**For more information, see page 101.**

### Functionality

Finally, you can write code to provide the dialogs and other functionality desired in your add-in.

### UNIX versus Windows

---

If you are developing add-ins for UNIX, you will run into the following differences:

**Table 14** *UNIX versus Windows*

<b>Windows</b>	<b>UNIX</b>
GUI painter and capture	GUI painter and capture
Custom dialogs	Custom dialogs
API through COM	MainWin—MainSoft Technology
Rose as COM client	MainWin—MainSoft Technology
ODBC functions	none
GUI drivers (for example, Send Keys)	n/a

The basic difference is that to create an add-in for UNIX, you must “fake” setting up the “registry”. When creating a UNIX version, follow these steps:

1. Copy the contents of `rose.version/addins/cm` directory to a new directory in the `rose.version/addins` directory where `version` is the installed version of Rational Rose.

For example:

```
cd rose.4.5.8153/addins
mkdir my_addin
cp cm/* my_addin
```

2. Change to the new directory

For example:

```
cd my_addin
```

---

**For more information, see page 101.**

3. Rename `cm.mnu` and `cm.reg` to the name of your add-in

For example:

```
mv cm.mnu my_addin.mnu
mv cm.reg my_addin.reg
```

4. Edit your menu file (`.mnu`) to add the menus you want  
The format is the same on Windows and UNIX
5. Edit the registry file (`.reg`) and replace “`cm`” with the name of your add-in. You should change:

- `HKEY_LOCAL_MACHINE`
- `InstallDir`
- `MenuFile`

A global search and replace on the document should help.

6. Copy your add-in’s custom help file to the `rose.version/help` directory where *version* is the installed version of Rational Rose.

For example:

```
cd rose.4.5.8153/help
cp /somepath/MyHelpFile.hlp .
cp /somepath/MyHelpFile.cnt .
```

## Creating portable add-ins

---

To create add-ins that will be portable to other platforms, keep the following recommendations in mind:

- Keep the logic of the integration in Rational Rose Script
- Keep dialogs and graphical user interfaces (GUIs) in Rational Rose Script
- Import and export through ASCII files
- Do not use COM calls—write shell-accessible commands
- Test the operating system with the BasicScript object (for example, Basic.OS)
- Use path map variables

---

**For more information, see page 101.**

### How to develop add-ins

---

The following procedure gives you a high-level look at what you need to do to develop your add-in:

1. Decide which language to use to create your add-in (Rational Rose Script or a COM-enabled language such as Visual Basic, C++).
2. Decide whether you will be a basic or language add-in.
3. Decide which parts of Rational Rose you want to customize or use:
  - ❑ Main menus
  - ❑ Shortcut menus
  - ❑ Custom Specifications
  - ❑ Properties
  - ❑ Data types
  - ❑ Stereotypes
  - ❑ Online help
  - ❑ Context-sensitive help
  - ❑ Registering for events
4. Design your add-in's functionality.
5. Create all the pieces for your add-in that you decided you needed:
  - ❑ Menu file (.mnu)
  - ❑ Property file (.pty) to add new tools, sets, and properties
  - ❑ Data types
  - ❑ Stereotypes (.ini, .bmp, .wmf, .emf)
  - ❑ Online and context-sensitive help (.hlp)
  - ❑ Method for updating the registry file (.reg)
  - ❑ Code to
    - perform all the functions of your add-in (.ebs, .ebx, .exe, .dll, etc.)
    - register for and handle events
    - create shortcut menu items
    - define your custom specification dialogs
  - ❑ Installation routine

---

**For more information, see page 101.**



- Uninstallation routine
  - Hardcopy documentation
  - Anything else specific to your needs
6. Test your add-in and its pieces
- Installation
  - Activation
  - New functionality
    - Menu items
    - Shortcut menu items
    - Custom Specifications
    - Properties
    - Data types
    - Stereotypes
    - Online help
    - Context-sensitive help
    - Events
    - All other add-in functionality
  - Deactivation
  - Uninstallation
7. Package your add-in and distribute to your customers (whether internal or external).

Working with and customizing each of the items listed above (for example, menus, properties) are explained in more detail in the next sections.

## Customizing Main Menus

Each add-in may introduce additions to the menus specific for that add-in, using the menu file technology (\*.mnu). This is the only way an add-in can provide main menu items.

For more information on the syntax for the Rational Rose menu file (\*.mnu), see the *Customizing Rational Rose Main Menus* section earlier in this User's Guide.

---

**For more information, see page 101.**

**Note:** *If you choose to customize the main menus, you must update the registry (discussed later in this appendix).*

### Customizing the Shortcut Menu

For information on customizing the shortcut menu, see the *Customizing Rational Rose Shortcut Menus* section earlier in this User's Guide.

### Creating Custom Specifications

To create and activate custom specifications, do the following:

1. Create a language add-in.
2. Register for the OnPropertySpecOpen event.
3. Implement an OnPropertySpecOpen interface in your add-in's OLE server.
4. Code your custom specification dialogs.

### Customizing Properties

Properties are added to Rational Rose items by add-ins using the existing property file (pty-file) technology. Each Add-in can optionally supply its own property file that defines a name space for its properties and a tab in the specification editor to hold the custom tool, sets, and properties. You can only define one property file per add-in, but you can define multiple tools, sets, and properties within that one file. The property file is automatically enabled and disabled as your add-in is enabled and disabled. Even when the property file is disabled, however, your custom properties are persisted with the model file. To hide a tab, the user can deactivate the corresponding add-in in Rational Rose.

### Design Considerations

The ordering of the tabs (tools) must be independent of when, where, and what add-ins are installed or activated. The tab name (tool name) must be unique for each Add-in. Rational Rose has no capability to detect conflicts. You must always have a "default" set for each of your custom tools.

---

**For more information, see page 101.**

---

**Note:** *If you choose to add a property file, you must update the registry (discussed later in this appendix).*

You can also add, delete, and clone properties through the extensibility interface. For more information on how to do this, see *Managing Default Properties* and subsequent sections earlier in this User's Guide.

## Information in Property Files

Property Files contain the following information:

```
version
  tool 1
    default set__model element 1
      property 1
      property 2
      ...
      property n
    default set__model element 2
      property 1
      property 2
      ...
      property n
    default set__last model element
      ...
  next set__model element 1
    property 1
    property 2
    ...
    property n
  next set__model element 2
    ...
  next set__last model element
    ...
last set__model element 1
```

---

**For more information, see page 101.**

```
...
    last set__last model element
...
tool 2
    default set__model element 1
...
    last set__last model element
...
last tool
...
```

### Format for Property Files

Now that you have seen an overview of the information contained in property files, it is time to look at their actual format. Keywords are shown in **bold**, while variable information, that you need to set, is shown in *italics*. Each element is explained at the end of the property file format.

```
# Comments about the property file
# Begin version information
(object Petal
  version    number
  _written   "add-in name"
  charSet    0)
# End version information

# Begin tool definition
(list Attribute_Set
  # Tool setup
  (object Attribute
    tool      "tool"
    name      "propertyID"
    value     "809135966")

  # Begin set and model element definition
  (object Attribute
    tool      "tool"
    name      "set__model element")
```

---

**For more information, see page 101.**

```
value (list Attribute_Set
      # Define first property
      (object Attribute
        tool "tool"
        name "property"
        value datatype)

      # Define second property
      (object Attribute
        tool "tool"
        name "property"
        value datatype)
      ...

      # Define nth property
      (object Attribute
        tool "tool"
        name "property"
        value datatype)
    )
  # End property list
)
# End set and model element list

# Begin next set and model element list
(
  ...
)
# End next set and model element list
)
# End tool definition

# Begin next tool definition. Repeat format.
(
  ...
)
# End next tool definition
# End property file
```

**For more information, see page 101.**

The property file is composed of the following elements:

- **comments:** Place a number sign (#) at the beginning of the line to indicate that it is a comment line.
- **number:** Enter the petal version number that corresponds to the version of Rational Rose for which you are writing your add-in. To find out what this number is, first locate a model file (.mdl) saved in the same version of Rational Rose. Next, open the model file in a text editor, such as Notepad.
- **add-in name:** Enter the name you want to call your add-in. For example, Rose/MyAddin v1.0
- **tool:** Enter the name of your tool. For example, My Tool. You may define multiple tools for your add-in in one property file.
- **value:** Use the same value (809135966) for each of your tools. If you run into problems, add 1 to the number.
- **set\_model element:** Enter the name of your set and model element. For example, default\_Project, CompilerV1.0\_Project, CompilerV2.0\_Project, default\_Class. You may have multiple sets and multiple model elements per tool. Valid model elements are:
  - Association
  - Attribute
  - Category
  - Class
  - Has
  - Inherit
  - Module-Spec
  - Module-Body
  - Operation
  - Param
  - Project
  - Role
  - Subsystem
  - Uses

---

**For more information, see page 101.**

- **property:** Enter the name of your property. For example, minCount
- **datatype:** Enter the default value for the data type of your property. For example, if your property is
  - an integer, your default value may be 0
  - a string, your default value may be "" or "Unknown"
  - a boolean, your default value may be TRUE

The following table lists examples for each of the different data types and how to format them in your property file. Note the cases where quotes are used versus where they are not used.

**Table 15 Property file data types**

<b>Data type</b>	<b>Example and Default</b>	<b>Format</b>
String	Name blank	(object Attribute tool "MyTool" name "Name" value "")
Integer	minCount 0	(object Attribute tool "myTool" name "minCount" value 0 )
Boolean	isRelated FALSE	(object Attribute tool "myTool" name "isRelated" value FALSE)

**For more information, see page 101.**

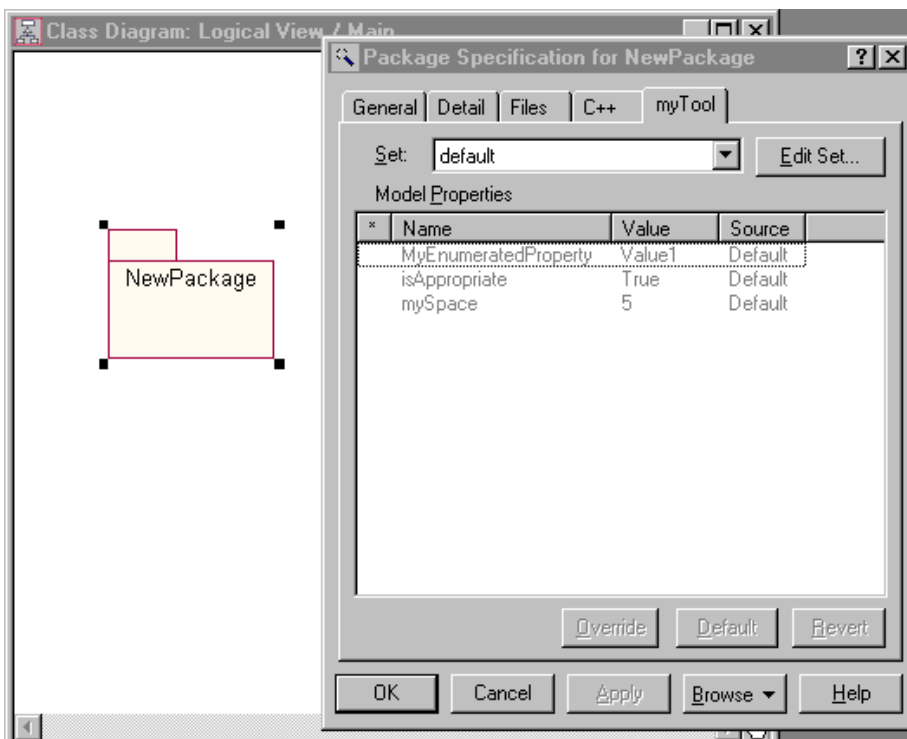
<b>Data type</b>	<b>Example and Default</b>	<b>Format</b>
Multi-line string	Description Blank	<pre>(object Attribute   tool  "myTool"   name  "Description"   value (value Text ""))</pre>
Enumeration (setup)	Color n/a	<pre>(object Attribute   tool  "myTool"   name  "Color"   value (list Attribute_Set     (object Attribute       tool  "myTool"       name  "Red"       value 100)     (object Attribute       tool  "myTool"       name  "Blue"       value 110)     (object Attribute       tool  "myTool"       name  "Green"       value 120)))</pre>
Enumeration (usage)	Shade "Red"	<pre>(object Attribute   tool  "myTool"   name  "Shade"   value ("Color" 100))</pre>

**For more information, see page 101.**



## Sample Property File

To add the tool, set, and properties (with default values) displayed in the following property dialog box,



**Figure 22 Sample Custom Properties**

we created the following property file:

```
(object Petal
  version 43)

(list Attribute_Set
  (object Attribute
    tool "myTool"
    name "default_Category"
    value (list Attribute_Set
```

**For more information, see page 101.**

```
(object Attribute
  tool "myTool"
  name "MyNewEnumeration"
  value (list Attribute_Set
    (object Attribute
      tool "myTool"
      name "Value1"
      value 1)
    (object Attribute
      tool "myTool"
      name "Value2"
      value 2)
    (object Attribute
      tool "myTool"
      name "Value3"
      value 3)))
(object Attribute
  tool "myTool"
  name "MyEnumeratedProperty"
  value ("MyNewEnumeration" 1))
(object Attribute
  tool "myTool"
  name "isAppropriate"
  value TRUE)
(object Attribute
  tool "myTool"
  name "mySpace"
  value 5))))
```

**Note:** This tool tab only displays on package specifications, since we only defined them for packages (`default__Category`). To display this tab for classes, duplicate the `default__Category` section and rename it to `default__Class`.

For more examples of property files, see the .pty files that come with Rational Rose.

---

**For more information, see page 101.**

## Creating Property Files

To create a property file, for inclusion with your add-in, do the following:

1. Create a new text file with extension .pty in a text editor or copy an existing .pty file.
2. Edit the property file (.pty) as desired. Use the explanations given previously and existing property files to guide you.

## Testing Property Files

1. Create and save a test model with all the model elements for which you added properties.
2. Add the new property file by following menu path: **Tools > Model Properties > Add** and selecting your property file (.pty).
3. Check the error log to make sure your model properties were all loaded okay. For example:

```
16:35:51| [Add Model Properties]
16:35:51| Adding model properties from file C:\Program
Files\Rational\Rose 2000e\my model properties.pty.
16:35:51| A total of 4 model properties have been added to
the original model.
```

4. Test your new properties by opening the specification for each affected model element. Look for:
  - A new tab or tabs with your tool name or names
  - Correct sets (default, plus any others) on each tool tab
  - Correct properties for each set
  - Correct default values for each property
  - Correct data types for each property. For example, click on an enumerated type to make sure that Rational Rose displays a drop down box that includes all the valid values for your enumeration.

---

**For more information, see page 101.**

### Customizing Data types

Add-ins may also choose to optionally supply a set of default data types to be presented to the user for typing attributes, parameters, etc., in Rational Rose specifications. These data types are defined in the registry setting called **FundamentalTypes**. For information on updating the **FundamentalTypes** registry setting, see the section in this appendix, *Updating the Registry*.

### Customizing Stereotypes

An Add-in may supply a set of stereotypes and an additional set of metafile icons to represent them. These stereotypes will be loaded and made available to Rational Rose when the Add-in is activated. Your custom stereotypes are added to Rational Rose's default set of stereotypes for the UML. Custom stereotypes do not replace standard ones. The location of your custom stereotypes is defined in the registry setting called **StereotypeCfgFile**. For information on updating the **StereotypeCfgFile** registry setting, see the section in this appendix, *Updating the Registry*.

You may provide icons for your stereotypes or text. Stereotypes are applicable to the following model elements:

- Association
- Attribute
- Class
- Component
- Component Package
- Connection
- Dependency
- Device
- Generalization
- Logical Package
- Operation
- Processor
- Use Case

---

**For more information, see page 101.**

- Use Case Package

You may create custom icons for one or more of the following:

- Diagram editor icons (.wmf, .emf) to display on diagrams
- Diagram toolbar icons (.bmp) to display on the toolbar buttons
- Browser list icons (.bmp) to display in the browser

**Note:** *You only need one bitmap file for your diagram toolbar icons and a separate bitmap file for all your custom browser icons. You do not need separate bitmap files for each of these icons. An index into the bitmap is used to indicate which bitmap goes with which stereotype.*

### Steps for Creating Add-In Stereotypes

1. Decide on the model element(s) and text stereotype name(s)
2. Decide which, if any, graphical representations you want to customize:
  - Editor
  - ToolBar
  - Browser
3. Define the stereotype in the stereotype INI file for the add-in
4. Create the custom icon graphics (.wmf, .emf, .bmp)

### General INI File Format

**Note:** *In addition to the description in the next pages, you can also find information on custom stereotypes and the stereotype configuration file in the Stereotypes chapter of the Rational Rose 2000e, Using Rose book and online help.*

The stereotype INI file contains the following information:

#### [General]

This section contains add-in specific settings such as the name of the add-in and whether it is a language add-in.

---

**For more information, see page 101.**

### [Stereotyped Items]

This section is like a table of contents for the stereotypes. It contains a list of stereotyped REI objects. For example, Class:Control, Component:DLL, Operation:Set.

### [REI Item:Stereotype name]

This section contains the settings for each stereotype, including any optional icon files and settings.

### Example:

#### [General]

ConfigurationName=Name

IsLanguageConfiguration=Yes or No

#### [Stereotyped Items]

REI item:Stereotype name

REI item:Stereotype name

...

[REI item:Stereotype name]

Item=REI item

Stereotype=Stereotype name

optional icon settings:

Metafile=&/model-element.wmf

SmallPaletteImages=&/palette\_icons.bmp

SmallPaletteIndex=Index

MediumPaletteImages=&/palette\_icons.bmp

MediumPaletteIndex=Index

ListImages=&/stereotypes.bmp

ListIndex=Index

...

[REI item:Stereotype name]

Item=REI item

Stereotype=Stereotype name

optional icon settings:

Metafile=&/model-element.wmf

SmallPaletteImages=&/palette\_icons.bmp

SmallPaletteIndex=Index

MediumPaletteImages=&/palette\_icons.bmp

MediumPaletteIndex=Index

ListImages=&/stereotype.bmp

ListIndex=Index

---

**For more information, see page 101.**

The stereotype INI file is composed of the following elements:

- **ConfigurationName:** This is the name of the add-in or name used for maintenance.
- **IsLanguageConfiguration:** Type **Yes** if your add-in is a language add-in. Otherwise, type **No**. This information conditionalizes stereotypes so that they only appear if the language of the model element in Rational Rose is the same as the **ConfigurationName** listed above.
- **Item:** The model element for which you are defining a stereotype.
- **Stereotype:** The text string stereotype. This is the text to be displayed between guillemets (<< >>).
- **Metafile:** The windows metafile (.wmf) or enhanced metafile (.emf) containing your diagram editor icon stereotype.
  - Windows Meta Files (wmf) may require additional extent settings
  - Enhanced Meta Files (emf) are preferred
- **SmallPaletteImages:** The bitmap file (.bmp) containing all your small icons for your toolbar buttons. This defines non-large icons (15 pixels high x 16n wide).
- **SmallPaletteIndex:** The integer number indicating the location in the bitmap file of the small toolbar button icon for this stereotype. This index starts with 1.
- **MediumPaletteImages:** The bitmap file (.bmp) containing all your medium icons for your toolbar buttons. This defines large icons (24 pixels high x 24n wide).
- **MediumPaletteIndex:** The integer number indicating the location in the bitmap file of the medium toolbar button icon for this stereotype. This index starts with 1.
- **ListImages:** The bitmap file (.bmp) containing all your custom browser icons.
  - device independent bitmaps
  - 16 high x 16n pixels wide
  - white background
  - use paint or bitmap editor
  - **&** is the installation directory

**For more information, see page 101.**

- **ListIndex:** The integer number indicating the location in the bitmap file of the custom browser icon for this stereotype. This index starts with 1.

The following sections focus on the different types of text and icon stereotypes you can create. You do not need separate files for each of these items; all text and icon information can go in one INI file.

### **Text-Only Stereotypes INI File**

No custom icons are included—only the text stereotypes

#### **Example:**

```
[Stereotyped Items]
Class:Interface
Component:DLL
Component:ActiveX
Component:Application

[Class:Interface]
Item=Class
Stereotype=Interface

[Component:DLL]
Item=Component
Stereotype=DLL

[Component:ActiveX]
Item=Component
Stereotype=ActiveX

[Component:Application]
Item=Component
Stereotype=Application
```

---

**For more information, see page 101.**



### **Custom Diagram Editor Icons INI File**

Metafile must be used in the optional icon settings section to define diagram icons.

#### **Example:**

```
[Class:Actor]
Item=Class
Stereotype=Actor
Metafile=&/Objectory/color/actor.wmf
SmallPaletteImages=&/Objectory/palette_icons.bmp
SmallPaletteIndex=1
MediumPaletteImages=&/Objectory/palette_icons.bmp
MediumPaletteIndex=2
ListImages=&/Objectory/list_icons.bmp
ListIndex=1
```

### **Custom Toolbar Button Icons INI File**

SmallPaletteImages, SmallPaletteIndex, MediumPaletteImages, and MediumPaletteIndex must be used in the optional icons settings section to define diagram palette icons.

#### **Example:**

```
[Class:Actor]
Item=Class
Stereotype=Actor
Metafile=&/Objectory/color/actor.wmf
SmallPaletteImages=&/Objectory/palette_icons.bmp
SmallPaletteIndex=1
MediumPaletteImages=&/Objectory/palette_icons.bmp
MediumPaletteIndex=2
ListImages=&/Objectory/list_icons.bmp
ListIndex=1
```

---

**For more information, see page 101.**

### Custom Browser List Icons INI File

The [General] section is needed. ListImages and ListIndex must be used in the optional icon settings section.

#### Example:

```
[General]
ConfigurationName=Oracle8
IsLanguageConfiguration=Yes

[Stereotyped Items]
Class:ObjectType
Class:ObjectTable
...

[Class:ObjectType]
Item=Class
Stereotype=ObjectType
ListImages=&/o8stereo.bmp
ListIndex=3

[Class:ObjectTable]
Item=Class
Stereotype=ObjectTable
ListImages=&/o8stereo.bmp
ListIndex=4
```

### Additional online help

Each add-in may introduce additions to the on-line help when installed or activated. These additions are activated when your add-in is activated.

On-line help should cover the capabilities of the installed add-in. Each add-in should have only one first level help book in the master table of contents. So, for example, your add-in should add a single book, for example "My AddIn". There may be many books under that book, but only one book should appear in the main Rational Rose help table of contents.

---

**For more information, see page 101.**

## Adding Online Help for Your Add-In

To be included in the Rational Rose Help, the add-in .hlp and .cnt files must reside in the same directory as the rest of the Rational Rose help.

- In Windows, the help directory is specified by the HelpFileDir general Rational Rose registry setting. The registry key for this setting is [HKEY\_LOCAL\_MACHINE\SOFTWARE\Rational Software\Rose\HelpFileDir].
- In UNIX, the help directory is fixed: /rose.*version*/help where *version* is the version of the currently installed Rational Rose (for example, /rose.4.5.8153/help).

In addition, you must include the add-in.cnt file in the roseu.cnt file. This is accomplished by doing the following:

1. Locate the Rational Rose help files.
2. Identify the roseX.cnt file you wish to use:
  - Roseu.cnt supports the UML help variant
  - Rosec.cnt supports the COM help variantYou may choose to maintain both if you refer to both the UML and COM notations when running Rational Rose.
3. Make a backup copy of the RoseX.cnt file.
4. Open the roseX.cnt file.
5. Add the following lines to the top of the roseX.cnt file along with the other Index and Link definitions:

```
:INDEX = title =filename.hlp  
:LINK filename.hlp
```

Add the following line at the bottom of the roseX.cnt file:

```
:INCLUDE filename.cnt
```

where *title* is the text that appears in the title bar of the Contents window of the add-in help file, *filename.hlp* is the name of the add-in help file, and *filename.cnt* is the name of the add-ins's contents file.

---

**For more information, see page 101.**

### Additional context-sensitive help

To be consistent with Rational Rose, you can include context-sensitive help in your add-in for your custom menu items, properties, and user interface (your add-in's dialogs, for example). Each context-sensitive help topic must have an A-Keyword defined for it. Since menu items and properties are Rational Rose features, we explain the format needed to connect your custom menu items and properties to your context-sensitive help. Create your A-Links for your context-sensitive help in your chosen help authoring tool.

#### Main Menu Items

For main menu items, added via the menu file (.mnu), the format for the A-Keyword is explained below.

For items on submenus:

*Menu, Submenu, menu item*

For items not on submenus (items located directly on Rational Rose main menus)

*Menu, menu item*

For example, Tools, MyAddIn, MyScript would be the alias for a context-sensitive help topic that explains the "MyScript" menu option on the "MyAddIn" submenu of the "Tools" menu (**Tools > MyAddIn > MyScript**).

Menu, submenu, and menu item names in the A-Keyword must include all punctuation. For example, if your menu path includes ellipsis:

**Tools > My Language > Project Specification...**

Your A-Keyword must also include ellipsis:

Tools, My Language, Project Specification...

**Note:** *There is no F1 help for intermediary submenus, only for menu items. So for the examples listed previously, there is no F1 help for "MyAddIn" or "My Language". If you have defined help topics and A-Keywords, however, there is F1 help for "MyScript" and "Project Specification...".*

---

**For more information, see page 101.**

## Model Properties

The format for the A-Keyword is:

*property (model element, tool)*

where *property* is the name of your custom property, *model element* is the name of the model element to which your property is applied, and *tool* is the name of your tab (tool) in the specification.

For example, isAppropriate (Category, myTool) would be the alias for a context-sensitive help topic that defines the “isAppropriate” property that applies to the myTool Packages.

## User manuals

You may supply your own soft or hard copy documentation, for your add-in, that covers its installation, use and limitations.

## Registering for events

You may register your add-in for Rational Rose’s events, thus triggering functionality in your add-in when that event occurs in Rational Rose. Since any number of add-ins may trigger on the same event, the order in which your add-in entry points are called, must be independent of when, where, and what add-ins are installed or activated.

Responses to events are usually coded as COM server interfaces, but some events can be mapped to Rational Rose Scripts. Events map to either an interface on your COM server or a Rational Rose Script, if the particular event allows Rational Rose Script.

Some events only apply to language add-ins (for example, OnGenerateCode, OnPropertySpecOpen).

Events fall into one of the following categories:

- Registry entry required
  - Interface events
  - Script events
- No registry entry needed, but an OLE server is required

Interface and Script Events are explained further in the next section.

---

**For more information, see page 101.**

### Interface Versus Script Events

Rational Rose's registry-required events can be implemented in one of two ways, interface or script. An *interface event* requires a registered COM server (.dll) that includes an interface, named the same as the event, to handle the event. A *script event* requires a script that can be executed (.ebx) to handle the event.

### What events are available?

General Events:

- Model-related: OnNewModel, OnCancelModel, OnCloseModel, OnOpenModel, OnSaveModel
- Model element-related: OnNewModelElement, OnModifiedModelElement, OnDeletedModelElement
- When the Rational Rose application is initialized: OnAppInit
- Add-In activation/deactivation: OnActivate, OnDeactivate

Code Generation-Related:

- Generating source code: OnGenerateCode
- Browsing source code: OnBrowseHeader, OnBrowseBody

GUI-related:

- Override the specification dialog box: OnPropertySpecOpen
- Extends the context menu: OnSelectedContextMenuItems, OnEnableContextMenuItems

Each of these events are described in detail in the *Rational Rose 2000e Extensibility Reference Manual* and online help along with warnings and precautions. The detailed descriptions also include a *Registry and Server Requirements* section explaining whether the event requires a registry entry or OLE server.

---

**For more information, see page 101.**

## How to add events to your add-in

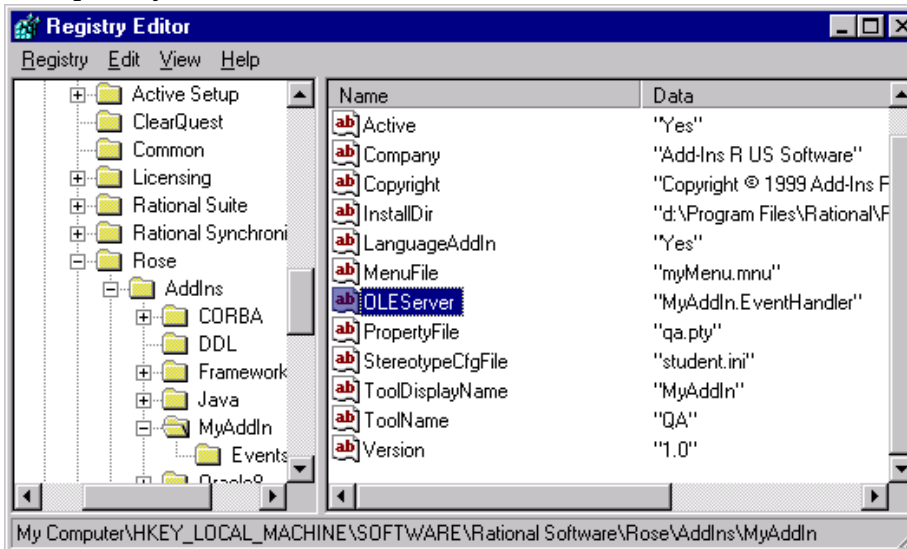
1. Add your COM server to your add-in registry
2. Add events to your add-in registry
3. Define an interface for each event
4. Register your COM server with the operating system

Each of these steps are detailed in the next sections.

### Step 1—Adding your COM server to the add-in registry

This step is optional for Rational Rose Script responses.

Set the **OLEServer** registry value to the name of your COM object (for example, `MyAddIn.EventHandler`):

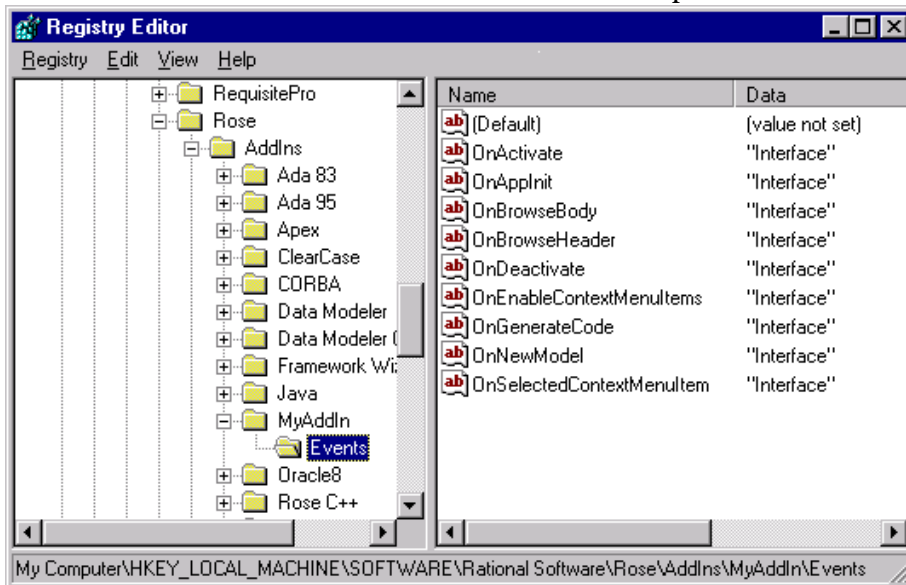


**Figure 23** OLEServer Windows Registry Entry

**For more information, see page 101.**

**Step 2—Adding events to the add-in registry**

1. Create an **Events** registry subkey under your add-in’s subkey.
2. List each event your add-in is registering for in the **Name** column.
3. Set the **Data** column to the Value as indicated:
  - “Interface” indicates a COM server call
  - “eventName.ebx” indicates Rational Rose Script execution



**Figure 24 Windows Registry Entries for Rational Rose Events**

**Note:** The script file must reside in the add-in’s installation directory, as specified by the add-in’s **InstallDir** registry setting, or a subdirectory of the add-in’s installation directory. If you choose to put the script in a subdirectory of your add-in’s installation directory (for example, \scripts), specify the subdirectory as part of the script file name (for example, \scripts\OnNewModel.ebx).

**For more information, see page 101.**

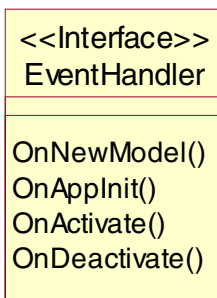


### Step 3—Defining an interface for each event

For each event for which your add-in is registering, name your interface or Rational Rose Script the same. For example, if your add-in is registering for the OnNewModel event, you would have one of the following:

- OnNewModel() interface in your OLE server

For example



**Figure 25** Sample Event Handler Defining an Add-In's Interfaces for Rational Rose Events

**Note:** Your COM server should only contain those events that you are responding to in your add-in.

- OnNewModel.ebx compiled Rational Rose Script

While the signature of the interface varies by event, most interface signatures are:

```
void event name (LPDispatch pRoseApp)
```

### Step 4—Register your COM server with the operating system

Add your COM server to the windows registry so that a client can get to it by COM object name (for example, CreateObject/GetObject). This is usually taken care of by the Integrated Development Environment (IDE). For example, Visual Basic registers your dll for you. Otherwise, to register your COM server, execute the command line:

```
regsvr32 file.dll
```

**For more information, see page 101.**

To verify that your COM server is registered, add a reference to it in the object browser of your IDE.

## Updating the Registry

Once the add-in is created, the following registry settings are necessary to enable an add-in. They are placed as sub-keys to a sub-key that represents the add-in name. The following would be an example of an add-in named MyAddIn:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Rational  
Software\Rose\AddIns\MyAddIn]
```

The add-in registry information should be placed in the Rose\AddIns folder of the registry.

## Registry Entries

The following registry entries are available when introducing add-ins.

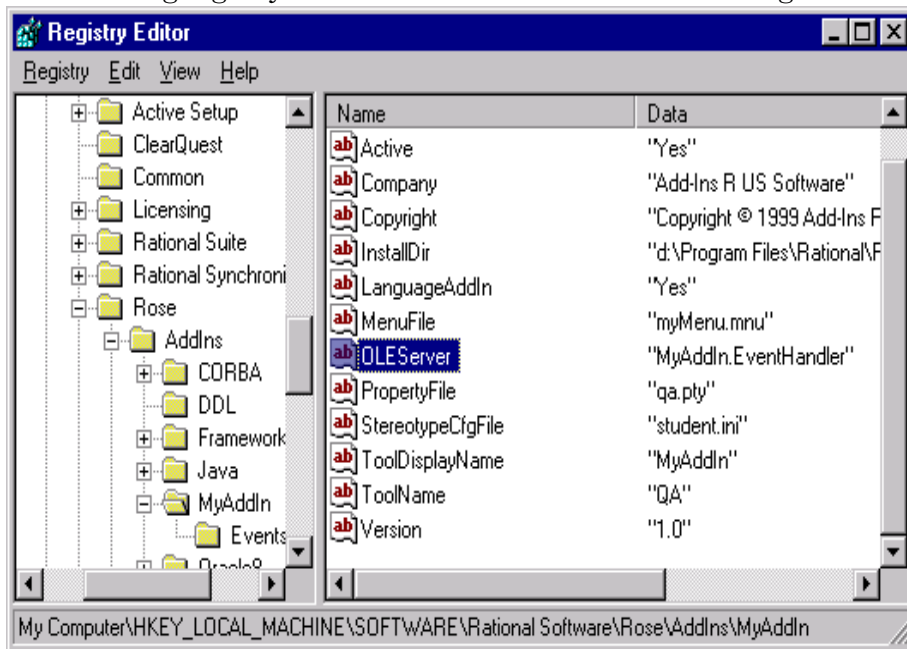


Figure 26 Windows Registry Entries for an Add-In

For more information, see page 101.

This list shows the registry sub-key names, descriptions, and defaults:

**Active**—Whether the Add-in is active or not. Settable by the user through the Rational Rose Add-in Manager. Default set to “Yes.”

**Company**—Name of the independent software vendor (ISV) that produced the add-in. For example, “Custom Software, Inc.”

**Copyright**—Specifies the copyright date of the add-in. For example, “©1996-1997”

**FundamentalTypes**—A String list of data types that appear in pull down menus for attributes when the add-in is active. This setting is required for all language add-ins. For example, “LOGICAL;CHAR;REAL” Note that this field is case sensitive.

**HelpFileName**—Name of the help file for the add-in, **without** any path or extension. For example, “myOnlineHelp”

**Note:** All add-in help files, including .cnt files, need to be located in the help directory specified by the **HelpFileDir** general Rational Rose registry setting (`[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose]`).

**InstallDir**—Directory where the add-in is installed. For example, “d:\My AddIn”

**LanguageAddIn**—Whether the add-in is a Round Trip Engineering (RTE) language add-in that wishes to use the component mapping feature. For example, “Yes”

**MenuFile**—Name of the menu file (\*.mnu) file that tailors Rational Rose. It needs to be installed in **InstallDir**. For example, “anaddin.mnu”

**OLEServer**—The name of the object that represents the OLE server that Rational Rose communicates with, if the add-in uses an OLE server. For example, “MyAddIn.EventHandler”

**Note:** The **OLEServer** value is case sensitive and only required if the add-in is using an OLE server to handle events.

**PropertyFile**—Name of the \*.pty property file for the Add-in for example, “user.pty”. This needs to be installed in **InstallDir**. This registry setting is required if the add-in is introducing properties.

---

**For more information, see page 101.**

**StereotypeCfgFile**—Specifies the custom stereotype configuration file for the add-in for example, “stereotypes.ini”. This setting is required if the add-in is introducing stereotypes. This needs to be installed in **InstallDir**.

**ToolDisplayName**—Specifies the add-in’s tool name that gets displayed on the properties tab and in the drop down list of languages in Rational Rose. This name can be different than the name that is used in the .pty file. Note that this is not a required setting. If this setting is not specified, the **ToolName** is displayed on the properties tab and in the drop down list of languages in Rational Rose. If this setting is specified, this is the name that gets assigned to a component. For example, “myLang” is the **ToolName** for the add-in, but “My Proprietary Language” is the **ToolDisplayName**.

**ToolList**—Displays the list of additional tools or property pages introduced by the add-in. Each tool is separated by a semi-colon. For example, “myLang;Tool2”. This setting is only required if the add-in introduces more than one property page.

**ToolName**—Specifies the add-in’s tool name, which must match the tool name in the add-in’s .pty file (the name that gets assigned to a component). For example, “myLang”, unless it’s overridden by a **ToolDisplayName**. In that case, the **ToolDisplayName** is assigned to the component.

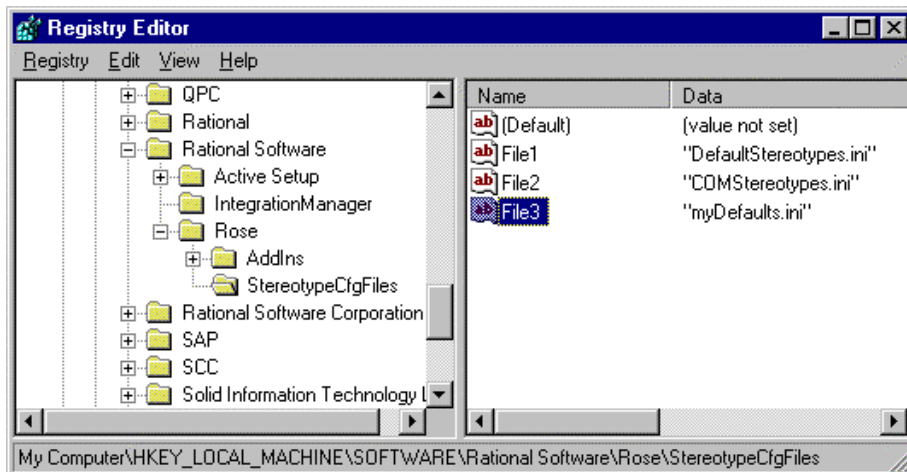
**Version**—Version number of the add-in, (not Rational Rose). For example, “1.2.3”

---

**For more information, see page 101.**

## Registering Custom Stereotypes

Add your stereotype INI file to the `StereotypeCfgFiles` subkey under the `Rose` Subkey. Name your entry `FileX` where `X` is the next available integer:



**Figure 27** Windows Registry Entry for an Add-In's Custom Stereotype Configuration File

The stereotype configuration file (.ini) must be located in the directory listed in the **InstallDir** registry setting.

## Updating the registry during installation

Since manual updates during installation are error-prone, we recommend that you avoid manual updates, like `regedit`. Instead, we suggest that you use an installation utility or execute a custom registry file.

### Installation utilities

Most installation utilities (for example, `InstallShield`) provide programmatic interfaces to the registry. Follow your installation utility's directions for updating the registry.

**For more information, see page 101.**

### Executing registry files

You can also update the registry by creating a registry file (.reg), then executing it during the installation of your add-in. To create a custom registry file, use one of the following methods:

- Create a registry file (.reg) from scratch in a text editor such as Notepad following the traditional INI file format, or
- Copy and edit an existing registry file (.reg), or
- Manually create your registry entries (in regedit, for example) then reverse engineer the format into a registry file (.reg):
  - select existing add-in registry settings in a registry editor
  - select the menu option to export the registry

### Registry File Anatomy

A registry file (.reg) looks like the following:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Rational
Software\Rose\AddIns\MyAddIn]
"Active"="Yes"
"Company"="Add-Ins R US Software"
"Copyright"="Copyright © 1999 Add-Ins R US Software Corp."
"LanguageAddIn"="Yes"
"Version"="1.0"
"PropertyFile"="qa.pty"
"MenuFile"="myMenu.mnu"
"StereotypeCfgFile"="student.ini"
"OLEServer"="MyAddIn.EventHandler"
"InstallDir"="d:\ProgramFiles\Rational\Rose 2000e\My AddIn"
"ToolName"="QA"
"ToolDisplayName"="MyAddIn"
...
```

---

**For more information, see page 101.**

## Installing, Setting up, and Uninstalling your add-in

After you finish designing and coding your add-in, it will consist of a combination of the following:

- Main menu items (.mnu)
- Shortcut menu items
- Custom specifications
- Properties (.pty)
- Data types
- Stereotypes (.ini, .bmp, .wmf, .emf)
- Online help (.hlp, .cnt)
- Context-sensitive help (.hlp)
- Event handling (.dll)
- Functionality through Rational Rose Scripts (.ebx) or controls (OLE-server) (.dll, .exe)
- Installation script
- Uninstall script

The purpose of the last two items, installation and uninstall scripts, is to introduce the files into the Rational Rose file structure and to register their locations, as well as other data needed by the framework, and to undo all this at a later time when the add-in is not wanted.

### Installation Reminders

When creating your installation script, remember the following:

- Install the pieces of your add-in (menu file, property file, etc.) in the subdirectory indicated in your add-ins InstallDir registry subkey.
- Update the roseX.cnt file as needed and install your help (.hlp) and contents (.cnt) file in the same directory as the Rational Rose help files.
- Update the windows registry, using your chosen method. Remember to do the following:
  - Create a registry subkey for your add-in (for example, [HKEY\_LOCAL\_MACHINE\SOFTWARE\Rational Software\Rose\AddIns\MyAddIn])

---

**For more information, see page 101.**

- ❑ Populate this subkey with the appropriate names and values (for example, InstallDir, MenuFile)
- ❑ If using events, create an Events subkey under your add-in subkey. Populate the events subkey with event names and “Interface” or “EventName.ebx” values.
- ❑ If using custom stereotypes, add your stereotype configuration file name (.ini) to the StereotypeCfgFile subkey.
- ❑ Create any other subkeys and registry entries needed for your implementation.

### Installing Add-Ins

It is possible for an add-in supplier to provide a programmatic and complete install of the add-in, for example, by using InstallShield. The same applies to reinstalls and updates. Installation changes will not take affect while Rational Rose is running, but will take affect upon start-up of Rational Rose.

Use the following steps to install an add-in on your Windows 95, Windows 98, or Windows NT system:

1. Exit Rational Rose.
2. Insert the application’s CD ROM or other media and run the **setup.exe** program.
3. Respond to the installation program’s dialogs to complete your installation.
4. Restart Rational Rose. Confirm that your add-in is installed and activated (checked) via the Rational Rose **Add-In Manager** menu.

### Uninstalling Add-Ins

We recommend that you provide a programmatic and complete uninstall of your add-in. Uninstall must remove not only the scripts, menu files, properties files, and help files, but must also clean the registry entries for the add-in.

---

**For more information, see page 101.**



## Activating and deactivating add-ins

Once an add-in is installed, it can be in an activated or deactivated state. Immediately after installation, new add-ins start out as activated.

When deactivated, an add-in is all but uninstalled:

- All menu items added by the add-in are removed.
- All property tabs added by the add-in disappear.
- All event bindings added by the add-in are disengaged.

**Note:** *A user may want to deactivate an add-in for a short time to keep it from functioning without actually uninstalling it.*

Add-ins are activated and deactivated in the Rational Rose user interface with the Add-In Manager. Add-ins are activated and deactivated programmatically with the REI AddInManager class.

---

**For more information, see page 101.**





## Index

### Symbols

# 114  
\$SCRIPT\_PATH 18  
%all\_units variable 13  
%current\_diagram variable 13  
%false variable 13  
%model variable 13  
%selected\_items variable 13  
%selected\_units variable 13  
%true variable 13  
%ufile variable 13  
%uname variable 13  
.bmp 121  
.emf 121  
.mnu 109  
.pty 110, 112  
.reg 138  
.wmf 121

### A

accelerator key, assigning 78  
accessing collections  
    by count 55  
    by name 56  
    by unique id 56  
activating, add-ins 141  
Active registry entry 135  
AddDefaultProperty method 45

Add-In Manager 5  
adding  
    comments to scripts 63  
    controls 85  
    menu entries 9  
    property to a property set 42  
    tools 53  
    virtual path for scripts 17  
add-ins  
    activating 141  
    active 135  
    A-Keywords 128  
    architecture 102  
    basic 103  
    COM servers 131  
    company name 135  
    contacting support 101  
    contents 103  
    context-sensitive help 105, 128  
    copyright 135  
    creating for UNIX 106  
    creating portable 107  
    data types 105, 120, 135  
    deactivating 141  
    developing 101, 108  
    events 105, 129, 130, 131, 132, 133  
    F1 128  
    functionality 106  
    help file name 135  
    inactive 135

- add-ins, *continued*
    - installation directory 135
    - installation utilities 137
    - installing 137, 139, 140
    - interface events 130
    - interfaces 133
    - language 103, 135
    - main menus 104, 109
    - manuals 129
    - menu file name 135
    - OLE server 135
    - online help 105, 126, 127
    - portable 107
    - properties 105, 110, 136
    - property files 111, 112, 135
    - Rational Unified Solutions Partner Program 101
    - registering COM servers 133
    - registry 134
    - script events 130
    - setting up 139
    - shortcut menus 21, 104, 110
    - specifications 104, 110
    - stereotypes 105, 120, 121, 136
    - support 101
    - technical support 101
    - tool list 136
    - tools 136
    - training 101
    - types 103
    - uninstalling 139, 140
    - UNIX 106
    - updating registry 137
    - user manuals 129
    - version 136
    - why create 103
  - adjusting attributes 89, 90
  - adjusting grid 76
  - A-Keywords
    - add-ins 128
    - properties 129
  - allfiles modifier 13
  - application object 40
  - assigning accelerator keys 78
  - Associating Files and URLs with Classes 40
  - Attribute 112
  - Attribute\_Set 112
  - attributes, adjusting 89, 90
  - automation controller 4
  - automation server 4
  - Automation, Rose 4
- ## B
- basename modifier 13
  - basic add-ins 103
  - BasicScript language 4
  - Block Action 11
  - Boolean 51
  - Braces 21
  - breakpoints
    - deleting manually 70
    - setting and removing 69
  - Browse Code 28
  - browser icons 120, 123, 124, 126
    - adding 105
- ## C
- Calls dialog box 68
  - capturing dialog boxes 78
  - Category.AddClass method 53
  - Category.RelocateClass method 53
  - changing
    - titles and labels 77
    - value of watch variable 73
    - write protection for a controllable unit 8
  - Char 51
  - Classes in Categories 53
  - clipboard, pasting from 63
  - CloneDefaultPropertySet method 47
  - Cloning a property set 47

- codefile modifier 14
  - collections
    - accessing by count 55
    - accessing by name 56
    - accessing by unique id 56
    - getting element from 55
  - COM servers 12, 129
    - registering 131, 133
  - comments
    - adding to script 63
    - property files 114
  - company name, add-ins 135
  - Company registry entry 135
  - compiling scripts 74
  - configuration files, stereotypes 121
  - ConfigurationName 123
  - context menus
    - See shortcut menus
  - context-sensitive help
    - add-ins 105, 128
    - main menus 128
    - properties 129
  - controllable units, working with 54
  - ControllableUnit.Control method 54
  - ControllableUnit.Load method 54
  - ControllableUnit.Save 54
  - ControllableUnit.SaveAs 54
  - ControllableUnit.Uncontrol method 54
  - ControllableUnit.Unload method 54
  - controller, automation 4
  - controls
    - adding 85
    - duplicating 86
    - incorporating in script 80
    - pastings into editor 88
    - repositioning 83
    - selecting 81
  - copying text 62
  - Copyright registry entry 135
  - copyright, add-ins 135
  - CreateDefaultPropertySet method 47
  - CreateProperty method 44, 45
  - creating
    - add-ins 103
    - custom specifications 110
    - new default property sets 41
    - new property 44
    - new property set 47
    - new property types 41
    - new scripts
      - from existing script 59
      - from scratch 58
    - new tools 41
    - portable add-ins 107
    - tool 53
    - UNIX add-ins 106
    - user-defined property type 51
  - current property set, getting and setting 49
  - Customized menus, capabilities of 8
  - customizing properties 110
  - customizing Rose menus, procedure 8
  - cutting text 62
- ## D
- data types
    - add-ins 105, 120, 135
    - customizing 105, 120
    - properties 115
    - property files 115
  - deactivating, add-ins 141
  - debugging a script 69, 70
  - default properties 5
  - DefaultModelProperties object 42
  - DeleteDefaultProperty method 44
  - DeleteDefaultPropertySet method 49
  - deleting
    - breakpoints manually 70
    - property 44
    - property set 49
    - text 62
    - watch variables 73
  - developing add-ins 101, 108

- diagram editor icons 120, 125
  - adding 105
- diagram toolbar icons 120
- Diagram.GetSelectedItems method 55
- Diagram.GetViewFrom method 54
- Diagram.Items method 54
- Diagram.ItemViews method 54
- Diagrams 3
- dialog box
  - capturing 78
  - editing 75
  - incorporating in script 80
  - inserting in script 75
  - past into editor 87
  - repositioning 83
  - selecting 81
  - testing 79
- dialog editor, working with 75
- dialogs
  - adding pictures to 86, 87
  - displaying information about 88
- directories
  - add-ins 135
  - installation 135
- directory modifier 14
- displaying
  - dialog for user input 8
  - grid 76
  - information about dialogs 88
- duplicating controls 86

## E

- editing dialogs 75
- elide modifier 14
- empty modifier 14
- Enumeration 51
- events
  - adding to your add-in 131
  - add-ins 105, 129
  - available 130
  - COM 105

## events, *continued*

- defining interfaces for 133
- details 130
- interface 130, 132
- main menus 12
- OnActivate 30, 130
- OnAppInit 130
- OnBrowseBody 130
- OnBrowseHeader 130
- OnCancelModel 130
- OnCloseModel 130
- OnDeactivate 130
- OnDeletedModelElement 130
- OnEnableContextMenuItems 30, 130
- OnGenerateCode 130
- OnModifiedModelElement 130
- OnNewModel 130
- OnNewModelElement 130
- OnOpenModel 130
- OnPropertySpecOpen 28, 130
- OnSaveModel 130
- OnSelectedContextMenuItem 31, 130
- registering 129, 132
- registering COM servers 133
- script 130, 132

## Exec Action 11

## executing

- program or shell script 8
- registry files 138
- Rose script 8

Extending Rational Rose course 101

extending Rose, ways to 1

Extensibility Components 2

## F

- F1, add-ins 128
- false modifier 14
- file modifier 14
- file names xix

- Files, associating with Classes 40
- finding
  - first itemview 54
  - procedure calls 68
  - text 64
- FindProperty method 45
- first modifier 14
- Float 51
- FundamentalTypes registry entry 120, 135
  
- G**
- Generate Code 28
- GetCurrentPropertySetName method 49
- GetPropertyValue method 45
- getting
  - current property set 49
  - element from collection 55
  - model properties 45
  - Rose Application object
    - Automation 40
    - Scripting 39
- grid, displaying and adjusting 76
  
- H**
- headerfile modifier 15
- help
  - context-sensitive 105
  - online 105
- help file name, add-ins 135
- HelpFileName registry entry 135
- home\_unit modifier 15
  
- I**
- icons
  - adding 105, 120
  - stereotypes 120
- InheritProperty method 44, 45
- inserting dialog in script 75
- insertion point, moving 59, 60
- installation directories, add-ins 135
- installation utilities 137
- InstallDir registry entry 135
- installing, add-ins 139, 140
- Integer 51
- interface events 130, 132
- InterfaceEvent action 12
- interfaces 129
  - accessing 12
  - defining for events 133
- interscript calls
  - debugging 75
  - guidelines for 74
- IsLanguageConfiguration 123
- Items 54
- itemview
  - currently selected 55
  - finding first 54
- ItemView.IsSelected method 55
- ItemViews 54
- iterating
  - through item views 54
  - through the items 54
  
- L**
- labels, changing 77
- language add-ins 23, 103
  - registry entry 135
- LanguageAddIn registry entry 135
- language-dependent 22
- language-neutral 23
- ListImages 123
- ListIndex 124
- Load or save controllable units 8
- Logical View of REI Model 2

### M

- main menus
  - add-ins 109
  - COM interfaces 12
  - context-sensitive help 128
  - customizing and extending 7, 104, 109
  - events 12
- managing default properties 41
- manuals, add-ins 129
- MediumPaletteImages 123
- MediumPaletteIndex 123
- Menu Action 11, 20
- Menu Argument 20
- Menu Entry 20
- menu files
  - Keywords 10
  - Modifiers 13
  - name 135
  - registry entry 135
  - syntax rules 15
  - Variables 13
- MenuFile registry entry 135
- menus
  - See also main menus
  - See also shortcut menus
- Metafile 123
- model elements 3
  - property files 114
  - stereotypes 123
- modifying value of watch variable 73
- multiple modifier 15
- multiple sets 5

### N

- non-language add-ins 24, 103
- not modifier 15

### O

- OLEServer registry entry 131, 135
- OnActivate event 30, 130
- OnAppInit event 130
- OnBrowseBody event 130
- OnBrowseHeader event 130
- OnCancelModel event 130
- OnCloseModel event 130
- OnDeactivate event 130
- OnDeletedModelElement event 130
- OnEnableContextMenuItems event 30, 130
- OnGenerateCode event 130
- online help
  - adding 105, 127
  - add-ins 126
  - REI xviii
- OnModifiedModelElement event 130
- OnNewModel event 130
- OnNewModelElement event 130
- OnOpenModel event 130
- OnPropertySpecOpen event 28, 110, 130
- OnSaveModel event 130
- OnSelectedContextMenuItem event 31, 130
- opening a script 58
- Option Entry 20
- Option Keyword 10
- organization of the manual xviii
- OverrideProperty method 44, 45

### P

- pasting
  - controls into editor 88
  - dialog box into editor 87
  - text from clipboard 63
- pausing an executing script 66
- petal version number, property files 114
- pictures, adding to dialog 86, 87
- portable add-ins 107



- prefix, Rose 53
- prerequisites xvii
- procedure calls, finding 68
- properties
  - adding to a property set 42
  - add-ins 110
  - A-Keywords 129
  - context-sensitive help 129
  - creating 44
  - customizing 105, 110
  - data types 115
  - default 5
  - defined 5
  - deleting 44
  - getting model 45
  - in add-ins 105
  - registry entry 136
  - sets 110
  - setting 45
  - tool display name 136
  - tool name 136
  - tool registry entry 136
  - tools 110
  - tools list 136
  - types 51
- property files 5
  - comments 114
  - creating 119
  - customizing 110
  - data types 115
  - design considerations 110
  - format 111, 112
  - model elements 114
  - petal version number 114
  - property name 115
  - registry entry 135
  - sample 117
  - sets 114
  - testing 119
  - tools 114
- property name, property files 115

- property sets
  - cloning 47
  - creating 47
  - deleting 49
  - multiple 5
- Property Specification Editor 41
- PropertyFile registry entry 135

## R

- Rational Rose Add-In Manager 5
- Rational Rose Application 2
- Rational Rose Application Components 2
- Rational Rose Automation 3, 4
  - syntax for 6
  - type libraries for 53
- Rational Rose diagrams, working with 54
- Rational Rose Extensibility Interface 3
- Rational Rose Extensibility Interface (REI) Model 1
- Rational Unified Solutions Partner Program 101
- readme.txt xix
- regedit 138
- registering for events 105, 129
- registry
  - installing 138
  - updating 134, 137
- registry entries 129
  - Active 135
  - Company 135
  - Copyright 135
  - FundamentalTypes 135
  - HelpFileName 135
  - InstallDir 135
  - LanguageAddIn 135
  - MenuFile 135
  - OLEServer 135
  - PropertyFile 135
  - StereotypeCfgFile 136
  - ToolDisplayName 136
  - ToolList 136

- registry entries, *continued*
    - ToolName 136
    - Version 136
  - registry files
    - executing 138
    - format 138
  - registry settings, exporting 138
  - regsvr32 133
  - REI Model
    - description 1
    - Logical View 2
  - release notes xix
  - removing breakpoints 70
  - replacing text 65
  - repositioning
    - controls 83
    - dialog boxes 83
    - items 82
  - resizing items 84
  - retrieving
    - all selected items 55
    - model properties 45
  - Rose menu file, sample 19
  - Rose Menus, Customizing 7
  - Rose prefix 53
  - Rose Script 3
  - Rose script editor 4
  - Rose Scripting language 4
  - Rose Scripts 129
  - Roseload Action 11
  - Rosesave Action 11
  - Rosescript Action 11
  - running a script 66
- S**
- sample
    - property file 117
    - Rose menu file 19
    - scripts 4
  - Script Editor, application window 57
  - script events 130, 132
  - Scripting language 4
  - scripts
    - sample 4
    - virtual path for 17
  - selecting
    - control 81
    - dialog boxes 81
    - text 61, 62
    - variables 72
  - Separator Entry 19
  - Separator Keyword 10
  - server, automation 4
  - SetCurrentPropertySetName method 50
  - sets
    - properties 110
    - property files 114
  - setting
    - current property set 49
    - model properties 45
      - Using InheritProperty 46
      - Using OverrideProperty 45
  - setting up, add-ins 139
  - shortcut menu items
    - adding 31
    - changing states 32
    - displaying 26
    - editing 31
    - formatting and displaying 24
    - retrieving 31
  - shortcut menus
    - activities 23
    - add-ins 104, 110
    - benefits 21
    - customizing 21, 29, 104, 110
    - decisions 23
    - deployment units 23
    - designing 28
    - diagrams 23
    - events 30, 31
    - extending 7, 104
    - external documents 23
    - formatting 26

- shortcut menus, *continued*
    - how it works 24
    - language-neutral model elements 23
    - limitations 22
    - menu items
      - adding 31
      - changing states 32
      - editing 31
      - formatting and displaying 24
      - retrieving 31
    - models 23
    - packages 23
    - properties 23
    - sample scripts 32, 34
    - scenarios 26
    - states 23
    - subsystems 23
    - swimlanes 23
    - synchronizations 23
    - transitions 23
    - use cases 23
  - SmallPaletteImages 123
  - SmallPaletteIndex 123
  - sourcefile modifier 15
  - specifications
    - add-ins 104, 110
    - custom, opening 28
    - customizing 104, 110
    - standard, opening 28
  - StepInto tool 67
  - StepOver tool 67
  - Stereotype 123
  - StereotypeCfgFile registry entry 120, 136
  - StereotypeCfgFiles 137
  - stereotypes
    - add-in type 123
    - add-ins 105
    - bitmap files 123, 124
    - browser icons 123, 124, 126
    - configuration file format 121
    - configuration name 123
    - creating 121
    - customizing 105, 120
    - diagram editor icons 123, 125
    - enhanced metafile 123
    - icons 105
    - model elements 123
    - registering 137
    - registry entry 136
    - text string 123
    - text-only 124
    - toolbar button icons 123, 125
    - windows metafile 123
  - stopping an executing script 66
  - String 51
  - Submenu Entry 20
  - Summit BasicScript language 4
  - syntax
    - in Rose Automation 6
    - in Rose Script 6
    - REI xvii
  - syntax rules, menu file 15
- ## T
- testing dialog boxes 79
  - text-only stereotypes 124
  - titles, changing 77
  - Tool, defined 5
  - toolbar button icons 123, 125
    - adding 105
    - stereotypes 123
  - ToolDisplayName registry entry 136
  - ToolList registry entry 136
  - ToolName registry entry 136
  - tools
    - display name 136
    - list 136
    - name 136
    - properties 110
    - property files 114
    - registry entry 136
  - tracing script execution 67

## Index

---

tracking variables 71  
type libraries 5, 53  
Type, defined 5  
type, property 51

## U

unary modifier 15  
uninstalling, add-ins 139, 140  
UNIX file names xix  
UNIX versus Windows, add-ins 106  
Updateaccess Action 12  
URL, associating with Classes 40  
user manuals, add-ins 129  
using type libraries 53  
utilities, installation 137

## V

Variables 13  
variables with modifiers, syntax 12  
variables, tracking 71  
Version registry entry 136  
virtual path for scripts 17

## W

watch list 72  
watch variables, adding 71  
working with controllable units 54  
working with Rose diagrams 54  
writable modifier 15