Using Rational PerformanceArchitect

support@rational.com
http://www.rational.com

Rational software

Using Rational PerformanceArchitect

Copyright © 1999 Rational Software Corporation. All rights reserved. The contents of this manual and the associated software are the property of Rational Software Corporation and are copyrighted. Any reproduction in whole or in part is strictly prohibited. For additional copies of this manual or software, please contact Rational Software Corporation.

Rational, the Rational logo, PerformanceStudio, SiteCheck, TestFactory, TestStudio, Object-Oriented Recording, and Object Testing are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies.

U.S. GOVERMENT RIGHTS. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational License Agreement and 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.

Revised 12/1999

This manual prepared by:
Rational Software Corporation
20 Maguire Road
Lexington, MA 02421
U.S.A.

Phone:
800-433-5444
408-863-4000

E-mail: support@rational.com
Web: http://www.rational.com

Contents

Road Map 2
Before You Begin 3
  Who Should Read This Guide 3
  What You Need to Get Started 3
Installing Rational PerformanceArchitect 4
  Before You Install 4
  Running the Installation 4
Basic Concepts 5
  Modeling Rational PerformanceStudio Features Using Rose . . . 5
  Generating Scripts and Wrapper Code from Rose Models 6
  Running LoadTest Schedules 7
  Understanding Datapools 8
Working with the Sample Model 8
  Understanding the Component View 8
  Understanding the Use Case View 10
  Understanding the Logical View 11
  Testing the Sample Model 12
Working with the Starting from Scratch Tutorial 17
Digging Deeper 17
  Adding the RationalTest Package to Your Models 18
  Converting Model Elements to a Virtual User Script 18
  Datapool Commands in the Virtual User Script 19
  Controlling Object Creation Overhead 22
  Modeling COM Objects with Rose 25
  How Rational PerformanceArchitect Maps VB Data Types . . . 29
  Sample VU Script 30
  Sample C++ Wrapper Generated for the VBExample Script . . 31
Troubleshooting 35
  Wrappers Fail to Build 35
  Asynchronous Environments 37

Runtime Registry Settings. 38

Using Rational PerformanceArchitect

Rational PerformanceArchitect (RPA) is a Rational Rose add-in that generates scripts for testing the performance of COM/DCOM applications. Rational PerformanceArchitect works by interpreting the information in Rose interaction diagrams and then generating scripts that can be used with Rational LoadTest. (Interaction diagrams include both sequence diagrams and collaboration diagrams.)

Rational PerformanceArchitect serves as a bridge between LoadTest and Rose. After you generate scripts in Rose, you can use the Robot component of PerformanceStudio to edit the scripts and then use LoadTest to run performance tests. Afterwards, you can rearchitect the model in Rose, generate new scripts, and compare the performance of the new architecture with that of the original.

Scripts generated by Rational PerformanceArchitect emulate the client side of a client/sever application, helping you catch design flaws in your architecture early in the development process, before you spend time and money implementing the client. All scripts generated by Rational PerformanceArchitect are virtual user scripts. Virtual user scripts are used primarily in performance testing to measure server response time. Virtual user scripts are written in a C-like scripting language called VU.

Rational PerformanceArchitect is extremely flexible and provides numerous usage scenarios for performance-based testing. In one scenario, you can model your COM/DCOM application for deployment in different languages and compare the performance of each version. In another scenario, you can compare the performance characteristics of a thin client with that of a fat client.

By default, Rational PerformanceArchitect generates one script per Rose diagram and assigns the script the same name as the diagram. As an alternative, you can build a diagram that generates multiple scripts. For an example of this approach, see *Controlling Object Creation Overhead* on page 22.

Road Map

Use the following table to help you find various topics in this guide.

To find out	Read this
Who should read this guide and what you need to get started	*Use the following table to help you find various topics in this guide.* on page 2
How to install Rational PerformanceArchitect	*Installing Rational PerformanceArchitect* on page 4
How to model PerformanceStudio features in Rational Rose	*Adding the RationalTest Package to Your Models* on page 18 *Converting Model Elements to a Virtual User Script* on page 18
How to generate scripts and wrapper code from Rose models	*Generating Scripts and Wrapper Code from Rose Models* on page 6
How to work with test data and use the datapool facility	*Understanding Datapools* on page 8 *Datapool Commands in the Virtual User Script* on page 19
How to work with the sample Rose model	*Working with the Sample Model* on page 8
How to generate multiple scripts from a single diagram	*Controlling Object Creation Overhead* on page 22
How to model interfaces	*Modeling COM Objects with Rose* on page 25
How to map Visual Basic data types to C++ data types	*How Rational PerformanceArchitect Maps VB Data Types* on page 29
How to troubleshoot Rational PerformanceArchitect	*Troubleshooting* on page 35
About registry settings used or modified by Rational PerformanceArchitect.	*Runtime Registry Settings* on page 38

Before You Begin

This section describes the intended audience for this guide and what you need to get started.

Who Should Read This Guide

This guide assumes you are a system architect or software developer who wants to generate performance tests from Rose models. To get the maximum benefit from this guide, you need to be familiar with the Unified Modeling Language, Rational Rose, Rational PerformanceStudio, and COM development. Although some concepts related to these products and technologies are covered in this guide, refer to the Rose and PerformanceStudio documentation for additional information.

What You Need to Get Started

To use Rational PerformanceArchitect, you need:

- Rational Suite PerformanceStudio 1.5 or later

  PerformanceStudio is a sophisticated tool for automating performance tests on client/server systems. PerformanceStudio includes Rational LoadTest, Rational Rose, Rational Robot, Rational Administrator, Rational TestManager, and Rational PerformanceArchitect, and is needed to run performance tests and to edit scripts.

- Rational Rose 2000 or later
- Microsoft Visual C++ Version 6.0

  Visual C++ Version 6.0 is needed to build the wrapper code that is called by the generated script. Wrappers serve as an interface between the script and the COM server.

- The Rational Test package for Rose

  The Rational Test package for Rose is a group of related actor classes that provide support for modeling runtime functions used by various Rational software components. The Rational Test package for Rose is included with Rational PerformanceArchitect.

Installing Rational PerformanceArchitect

This section explains how to set up your test environment and install Rational PerformanceArchitect.

Before You Install

Before you install Rational PerformanceArchitect, keep the following requirements in mind:

- At minimum, you need Rational Robot and Rational Rose installed on the computer running Rational PerformanceArchitect. The rest of PerformanceStudio can reside on the same computer or a different computer.
- To generate wrapper code automatically, Visual C++ must be installed on the same computer that is running Rational PerformanceArchitect.

Running the Installation

To install Rational PerformanceArchitect, run the typical Rational Suite PerformanceStudio installation. You can also install Rational PerformanceArchitect from the Windows Start menu after performing a custom installation of Rational Suite PerformanceStudio.

The installation adds an Rpa subdirectory to the Rose root directory and several files within the new directory, including:

File or Directory	Description
IslandHopperA.mdl	Rose model derived by reverse engineerng the Island Hopper sample application from Microsoft Visual Studio 6.
RationalTest.cat	Rose catalog containing the Rational Test package. This package contains several actors that provide support for modeling runtime functions, such as think time, timers, and datapool fetch. (In the UML, an actor is someone or something outside the system that interacts with the system.)
Sample.VB	Directory containing the Visual Basic ActiveX server sample
Sample.VC	Directory containing the Visual C++ ActiveX server sample
Readme.htm	Rational PerformanceArchitect release notes
Rpa.htm	Rational PerformanceArchitect online manual in HTML format

Basic Concepts

This section introduces several concepts that will help provide a foundation for your work with Rational PerformanceArchitect.

Modeling Rational PerformanceStudio Features Using Rose

Rational PerformanceArchitect uses Rose interaction (sequence and collaboration) diagrams to access PerformanceStudio features, such as think time, timers, and datapools. These features are made accessible to Rose via the RationalTest package, a collection of Rational-supplied actor classes that provide support for modeling runtime functions used by various Rational software components. The RationalTest package contains the following actors:

VU — Provides support for modeling runtime services such as the average think time, COM initialization and deinitialization, and the VU language INFO SERVER command.

Timer — Provides support for modeling arbitrary start and stop timer operations (`start_time` and `stop_time` functions)

Datapool — Provides support for modeling datapool runtime operations, such as `open` and `fetch`.

VuServices — Inherits all operations of the VU, Datapool, and Timer actors. You can use VuServices as a single actor in an interaction diagram in place of the other three actors.

Script — Provides support for generating multiple scripts from a single diagram.

Virtual User — Provides support for modeling the client-side of a transaction in an interaction diagram. Use the Virtual User actor as a placeholder for the actual client object. Because this actor class is not part of the actual production application, you can always add it to existing models without effecting the model.

The RationalTest package is included with the sample model that comes with the Rational PerformanceArchitect installation, but it must be added to your own models. For more information, see *Adding the RationalTest Package to Your Models* on page 18.

The following table shows how Rational PerformanceArchitect converts the features in the RationalTest package to commands in a virtual user script:

RPA converts this feature in a diagram	To this command in a VU script
think()	set Think_avg= nnn
fetch()	datapool_fetch(x)
open()	datapool_open("x")
start()	start_time
stop()	stop_time
vu_CoInitialize()	vu_CoInitialize()
vu_CoUninitialize()	vu_CoUninitialize()
infoserver()	INFO SERVER label= addr[,label= addr...]

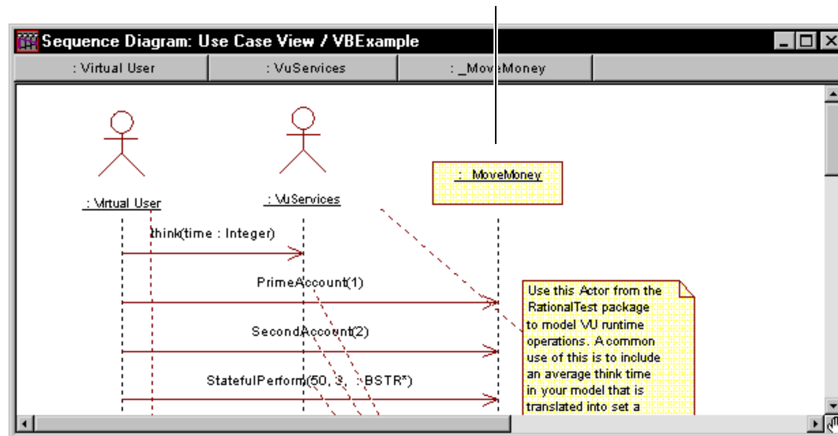
For more information about these features, see the *VU Language Reference*.

Generating Scripts and Wrapper Code from Rose Models

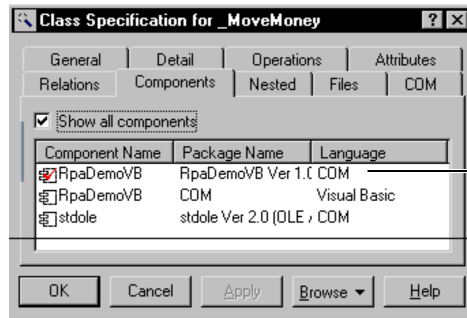
To generate a script, Rational PerformanceArchitect interprets the messages in an interaction diagram in your Rose model and then processes the operations associated with these messages. Specifically, Rational PerformanceArchitect inspects the class of the message's receiver object to determine whether the object is a COM object.

The following figure shows a message's receiver object in a Rose sequence diagram:

Message's receiver object in sequence diagram



If Rational PerformanceArchitect determines that the object is a COM object, it incorporates the message in the generated script as a VU `emulate` command. (See *Qualifying Messages for Inclusion in a Script* on page 27 for more information.) Ultimately, these messages, which are now incorporated in the script, help to drive a performance test.



You can inspect an object's Class Specification in Rose to see whether it is a COM object.

As it creates the script, Rational PerformanceArchitect also generates the C++ wrappers that are needed to establish an interface from the VU language to COM. Together, the VU script and wrappers emulate your client application using the actual COM calls of the server objects.

Rational PerformanceArchitect looks in the Component Specification in the Rose Component View for two pieces of information: the `FileName` property and the component name. Both pieces of information are saved in the wrapper and are shown in the following example from a wrapper file:

FileName property, displayed in the COM tab of the Component Specification.

```
#import "D:\Program Files\Rational\Rose
2000\RPA\Sample.vb\RpaDemoVB.dll"
```

Component Name

Running LoadTest Schedules

After you generate a script, you can add it to a LoadTest schedule and run the schedule. Typically, you add some number of virtual users to more accurately simulate real-world conditions. A virtual user is a single instance of a virtual user script running on a computer. Based on the results of the schedule run, you can choose to update the model in Rose, regenerate the scripts, and rerun the performance tests.

For more information about LoadTest, see the *Using Rational LoadTest* manual.

Understanding Datapools

With Rational Suite PerformanceStudio, you can create scripts that simulate the actions of multiple users running multiple transactions against a database or Web server. This is accomplished through the use of datapools. A datapool is a comma-separated text file that contains rows of data.

Scripts generated by Rational PerformanceArchitect are fully integrated with PerformanceStudio's datapool facilities. For more information about the datapool features of PerformanceStudio, see the *Using Rational LoadTest* manual. For more information about the use of datapools in Rational PerformanceArchitect, see *Datapool Commands in the Virtual User Script* on page 19.

Working with the Sample Model

The installation procedure installs two sample Rose models — a Visual Basic model and a Visual C++ model that you can use to try out Rational PerformanceArchitect. Both models reflect a traditional transaction processing system that allows users to credit or debit their accounts on a server.

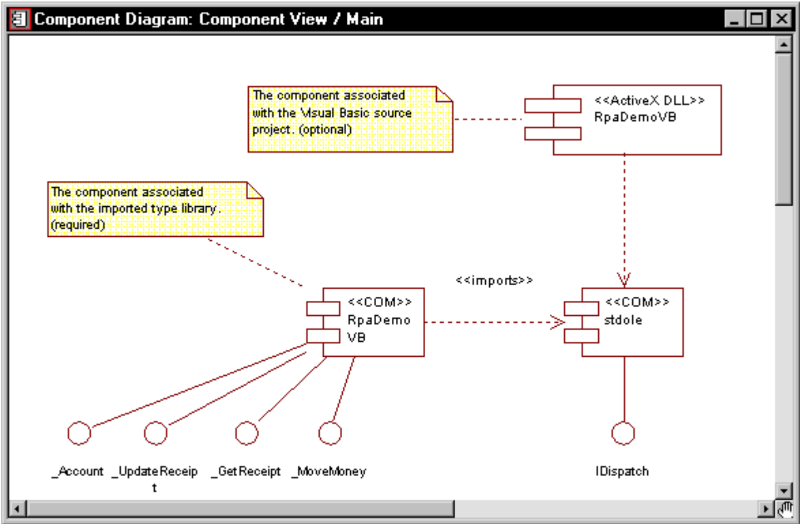
Both models already contain an imported type library and a component associated with the source project in Visual Basic and Visual C++.

Understanding the Component View

The following component diagram shows the physical pieces of software that are included in the architecture for the Visual Basic version of the sample banking application:

- A COM object (RpaDemoVB) that is associated with the imported type library
- An optional ActiveX DLL (RpaDemoVB) that is associated with the Visual Basic source project. This DLL is used for round-trip engineering.
- Stdole.dll, which is the COM automation library referenced by the RpaDemoVB project. This component is imported by Rose when the RpaDemoVB type library is imported.

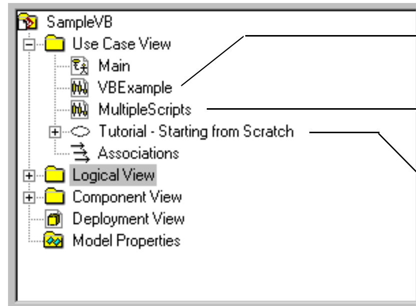
The client application uses the COM interfaces _Account, _UpdateReceipt, _GetReceipt, and _MoveMoney to communicate with the RpaDemoVB server. (All COM interfaces from imported Visual Basic applications contain a leading underscore character.)



NOTE: In the C++ version of the sample model, the COM object and the ActiveX DLL are combined into a single component.

Understanding the Use Case View

The Use Case view in the Rose browser shows the interaction diagrams and use cases that are included in the sample model.

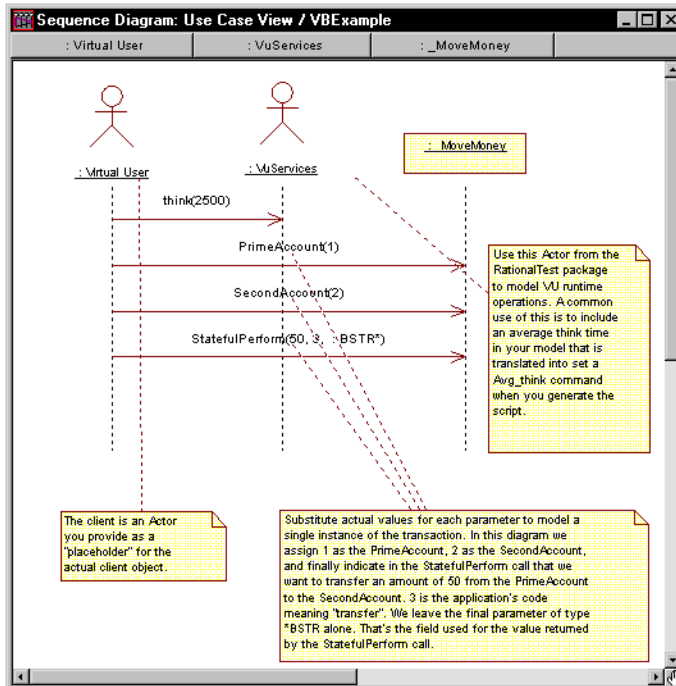


The sequence diagram used in this section of this document to help you test the sample model.

A 2nd sequence diagram that you can use to generate multiple scripts. (See Controlling Object Creation Overhead on page 22 for details.)

A tutorial that shows you how to generate a virtual user script from scratch. This use case includes multiple sequence diagrams.

The VBExample sequence diagram appears as follows. See *Controlling Object Creation Overhead* on page 22 for a description of the MultipleScript sequence diagram. See *Working with the Starting from Scratch Tutorial* on page 17 for a description of the other sequence diagram included with the sample model.



In the VBExample sequence diagram:

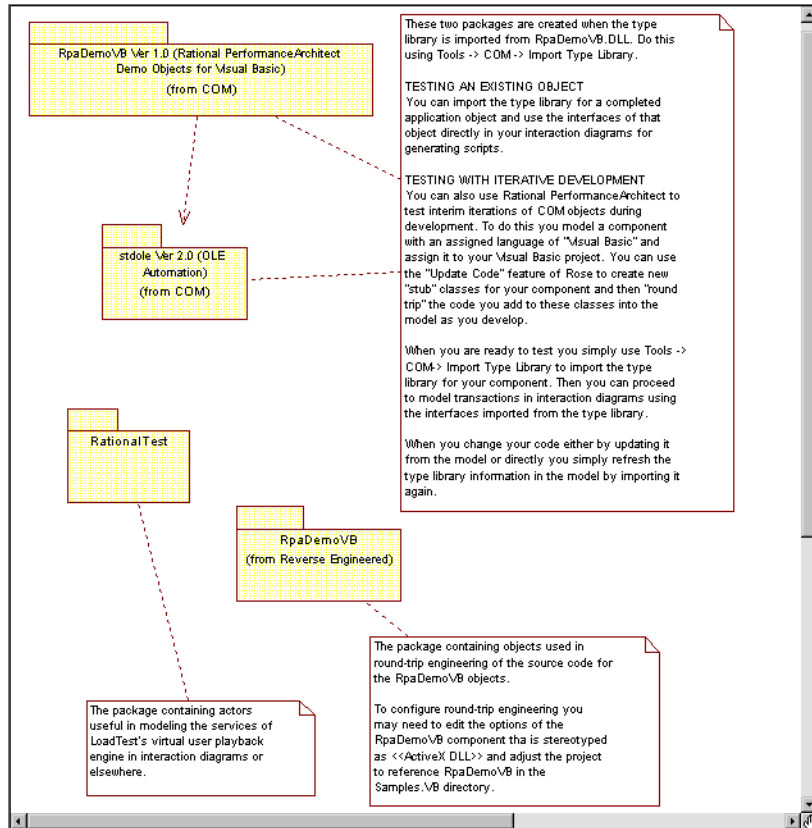
- The Virtual User actor serves as a placeholder for the actual client object.
- The VU Services actor enables runtime services such as think time.
- _MoveMoney is a hidden interface — indicated by the leading underscore character — and realized by the MoveMoney coclass. The _MoveMoney interface and the MoveMoney coclass are realized by the RpaDemoVB COM component.
- PrimeAccount assigns itself the number 1.
- SecondAccount assigns itself the number 2.
- StatefulPerform indicates a request to transfer $50 from PrimeAccount to SecondAccount.

Understanding the Logical View

The Logical View shows the packages, classes, interfaces, and operations in the model. Packages in the Visual Basic version of the sample model include:

- RpaDemoVB (from COM). The package that would be created when you import the type library from RpaDemoVB.dll. (In the sample model, the type library has already been imported.)
- stdole (from COM). The OLE Automation package that is created when you import a Visual Basic project.
- RationalTest. Rational-supplied actor classes that provide support for modeling runtime functions used by various Rational software components.
- RpaDemoVB (from Reverse Engineered). The package containing objects used in round-trip engineering of the source code for RpaDemoVB objects.

The following figure shows the main Class Diagram for the SampleVB model:



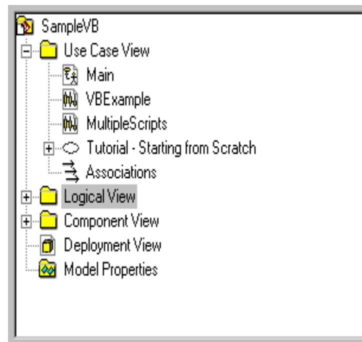
Testing the Sample Model

To see how Rational PerformanceArchitect works, you can generate a script and C++ wrappers from the VBExample sequence diagram. Then, you can use LoadTest to run a performance test. In the performance test, you can measure the time required for the `PrimeAccount`, `SecondAccount`, and `StatefulPerform()` messages that the client sends to the server.

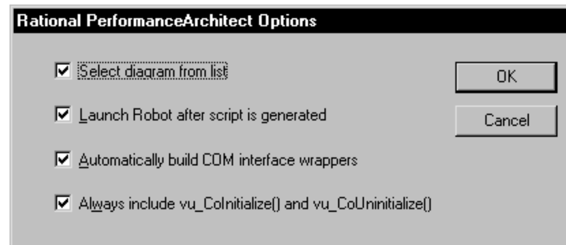
Generating the Script and Wrappers in Rose

To generate a virtual user script and the wrappers:

1. Start Rational Rose and open the sample model, [Rose Dir]\Rpa\SampleVB.mdl.
2. In the Rose Browser, expand the Use Case View.



3. Double-click the VBExample sequence diagram to open it.
4. Click Tools > VU Scripting > Options.



Make sure that all of the options are selected.

Select diagram from list — Displays a list of sequence diagrams to choose from. If deselected, Rational PerformanceArchitect generates a script based on the active sequence diagram.

Launch Robot after script is generated — Starts Robot automatically after generating the script.

> NOTE: If Robot is already running, newly generated scripts will not open automatically in Robot.

Automatically build COM interface wrappers — Builds the COM wrappers automatically. Rational PerformanceArchitect looks for the Microsoft Visual C++ 6.0 compiler on your computer. If it cannot locate the compiler, it asks you to locate the vcvars32.bat file, which is typically located in the VC98\bin directory of the compiler. If the compiler is not installed on your system, Rational PerformanceArchitect disables the Build wrappers automatically option. For more information about the wrappers, see *Generating Scripts and Wrapper Code from Rose Models* on page 6.

> NOTE: Deselect this option only in certain situations, such as when the C++ compiler does not reside on your computer. To build the wrappers manually, run the rpa.bat file, which resides in the repository under Project\[ProjectName]\Script\externC\[DiagramName].

Automatically include CoInitialize() and CoUninitialize() — Inserts a single `Vu_CoInitialize()` and a single `Vu_CoUninitialize()` statement in the script. These statements call the `CoInitialize()` and `CoUninitialize()` COM runtime functions. If deselected, these statements are not inserted into the script automatically, and therefore, these functions are not executed for each iteration of the script. These statements are part of the VU class in the RationalTest package. For more information about this option, see *Controlling Object Creation Overhead* on page 22.

5. Click Tools > VU Scripting > Generate Script.
6. When prompted, log into a Rational repository. (If you don't already have a repository, you will need to create one. For details, see the *Using the Rational Administrator* manual or the Rational Administrator online Help.)

Type your user ID and password. If you do not know these, see your administrator.

Select a repository. To change repositories later, exit all Robot components and log in again. (Repositories are created in the Rational Administrator.)

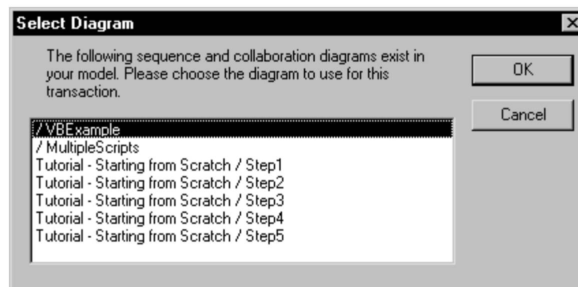
Select a project. You can change to another project within the same repository after you log in.

*Click **OK** to log in.*

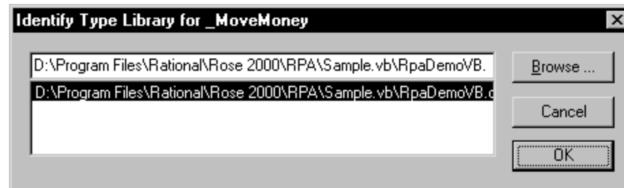
> NOTE: If your repository is located on another computer, you must map the path to the repository to a drive letter in order to use the Automatically build COM interface wrappers option.

7. If you have chosen the Select diagram from list option and your model contains multiple sequence diagrams, Rational PerformanceArchitect prompts you to select a sequence diagram. Select the VBExample sequence diagram and click OK.

> NOTE: The list can include both sequence and collaboration diagrams.



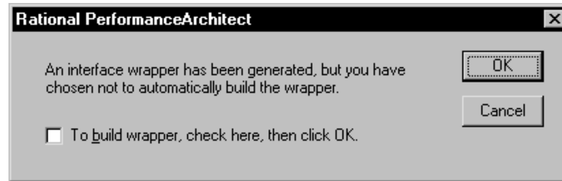
8. If you are prompted, type or browse to the path for RpaDemoVB.dll, which is the server component you are testing. Then, click OK.



> NOTE: Rational PerformanceArchitect looks for the path in the `FileName` property of the RpaDemoVB Component Specification. If it finds the path and the path is correct, you will not be prompted.

At this point, Rational PerformanceArchitect opens the script in Robot and builds the C++ wrapper DLL for the transaction described in the sequence diagram. The script is assigned the same name as the sequence diagram. For an example of a C++ wrapper, see *Sample C++ Wrapper Generated for the VBExample Script* on page 31.

If Build wrappers automatically is deselected in the Rational PerformanceArchitect Options dialog box, the following dialog box will appear at the end of the script/wrapper generation process. Enable the check box and click OK to build the wrappers.



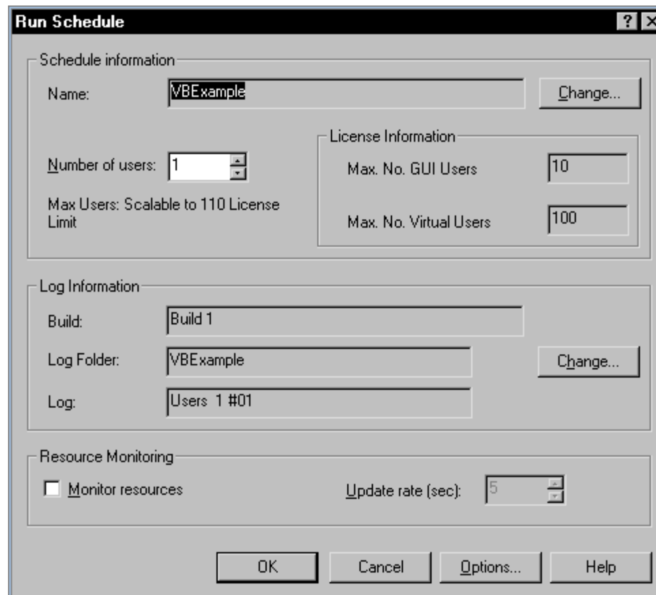
Running Performance Tests

To run a performance test:

1. In Robot, click File > Playback.

   This starts LoadTest and creates a LoadTest schedule from the virtual user script.

2. Click Run > Schedule.



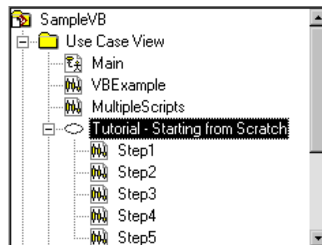
3. Click OK.
4. Review the results.

For more information about creating and running schedules, see the *Using Rational LoadTest* manual.

5. Run a Performance report in LoadTest to display the response times observed during the schedule run.

Working with the Starting from Scratch Tutorial

The Starting from Scratch tutorial is a use case that is included with the Visual Basic version of the sample model. This use case teaches you how to create your own sequence diagrams in which you model interface operations and generate a VU script.



Digging Deeper

Read the following topics in this section for important information about:

- Adding the RationalTest package to your models
- Converting model elements to a virtual user script
- Datapool commands in the virtual user script
- Controlling object creation overhead
- Modeling COM objects with Rose
- How Rational PerformanceArchitect maps Visual Basic data types
- Sample virtual user script
- Sample C++ wrapper

Adding the RationalTest Package to Your Models

The RationalTest package is a collection of actors that provide support for modeling runtime functions used by various Rational software components. To add this package to your own models:

1. Open your model in Rose.
2. Click **File** > **Units** > **Load.**
3. Navigate to the [Rose Dir]\rpa directory and select the **RationalTest.cat** file.
4. Click **Open**.

Converting Model Elements to a Virtual User Script

Rational PerformanceArchitect converts elements in your Rose model to lines of code in a virtual user script. To get a feel for how this conversion works, consider how COM calls and think time are modeled in your sequence diagram.

When Rational PerformanceArchitect generates a virtual user script, the messages to a COM server in the sequence diagram become VU language `emulate` commands in your script. The `emulate` command provides generic emulation services to the VU language and to external C function calls.

Rational PerformanceArchitect helps you model think time by providing a `think()` operation as part of the VU class in the RationalTest package. **Think time** is used in LoadTest to pace the playback of virtual user scripts. Generally speaking, think time is used to indicate the time that a typical user would delay or think between submitting commands.

The `think()` operation of the VU class in the RationalTest package takes one parameter, the average think time in milliseconds, for which you must supply a value.

Every `think` (*nnn*) message in your sequence diagram becomes a `set Think_Avg=nnn` command in the script, where *nnn* is the value of the average delay time in milliseconds. When you model this operation in a message, Rational PerformanceArchitect uses the value that you supply as the parameter on the message and generates the corresponding `set Think_avg` command in the virtual user script.

The following segment of a virtual user script shows how Rational PerformanceArchitect converts a `think` (*nnn*) message in the sequence diagram to a `set Think_Avg=nnn` command in the script and then converts the messages in the sequence diagram to `emulate` commands in the script.

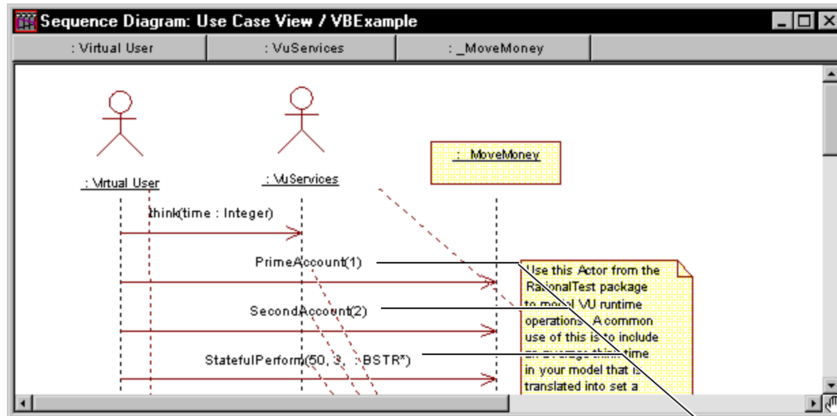
```
/* think() message from sequence diagram */
        set Think_avg = 2000;
        emulate ["IAccount_Post"] iRetval = IAccount_Post(
                        datapool_value(Transfer1VC, "lAccount"),
                        datapool_value(Transfer1VC, "lAmount")),
                        pszLogPass, pszLogFail;
```

For more information about the `emulate` command and the think time environment variable, see the *VU Language Reference*.

Datapool Commands in the Virtual User Script

Rational PerformanceArchitect uses the arguments from the COM calls in your interaction diagram to fill in a DATAPOOL_CONFIG section in the generated script. Like-named arguments with the same data value in the diagram are treated as one datapool field in the DATAPOOL_CONFIG section. Like-named arguments with different data values in the diagram are treated as separate fields in the DATAPOOL_CONFIG section. These rules apply even if the arguments are used in separate functions. To prevent like-named fields in the DATAPOOL_CONFIG section, Rational PerformanceArchitect appends a suffix to the argument name — for example AccountNo, AccountNo1, and so on.

In the following example, the AccountNumber parameter is assigned different values in the transaction while Amount is a constant value. This results in two datapool variables for AccountNumber and a single variable for Amount.



Data values shown in the COM calls in the sequence diagram become default values in the script.

```
DATAPOOL_CONFIG "VBExample" OVERRIDE  DP_SEQUENTIAL DP_SHARED DP_NOWRAP
{
        EXCLUDE, "lAccount", "string", "1";
        EXCLUDE, "lAccount0", "string", "2";
        EXCLUDE, "lAmount", "string", "50";
        EXCLUDE, "lTranType", "string", "3";
```

Defaults to Exclude

Data types appear but are not used by Rational PerformanceArchitect.

Column names come from the parameters in the Operation Specification in Rose.

This first line of the DATAPOOL_CONFIG contains the datapool name and the flags that define how the datapool is accessed when the script is played back in LoadTest.

Each subsequent line has four columns of information, separated by commas. These lines serve as a datapool blueprint, giving Robot the information it needs to create the datapool. During script playback, these lines also tell LoadTest where to look for values to assign the variables in the script.

You will find datapool commands throughout a virtual user script. These commands work in conjunction with the DATAPOOL_CONFIG section of the script to control datapool creation and datapool access. Datapool commands typically found in virtual user scripts include datapool_open, datapool_fetch, datapool_value, datapool_rewind, datapool_close, and DATAPOOL_CONFIG.

The following example shows several datapool commands and a `DATAPOOL_CONFIG` statement for a datapool named `VBExample`.

For more information about datapools, see the *Using Rational LoadTest* and *VU Language Reference* manuals.

```
 vu_CoInitialize();  /* Automatically generated - vu_CoInitialize() */
        VBExample = datapool_open("VBExample");
        datapool_fetch(VBExample);
       /* ******************************************************************
*/
       /* Operations mapped in the sequence diagram                        */
       /* ******************************************************************
*/
        /* think() message from sequence diagram */
        set Think_avg = 2500;
        emulate ["MoveMoney_PrimeAccount"] iRetval = MoveMoney_PrimeAccount(
                        datapool_value(VBExample, "lAccount")),
                        pszLogPass, pszLogFail;
        emulate ["MoveMoney_SecondAccount"] iRetval = MoveMoney_SecondAccount(
                        datapool_value(VBExample, "lAccount0")),
                        pszLogPass, pszLogFail;
       emulate ["MoveMoney_StatefulPerform"] iRetval = MoveMoney_StatefulPerform(
                        datapool_value(VBExample, "lAmount"),
                        datapool_value(VBExample, "lTranType")),
                        pszLogPass, pszLogFail;

       /* ******************************************************************
*/
        vu_CoUninitialize(); /* Automatically generated - CoUninitialize() */
        pop [Think_avg, Think_dist, Think_def, Timeout_val, Timeout_scale];
}
DATAPOOL_CONFIG "VBExample" OVERRIDE DP_SEQUENTIAL DP_SHARED DP_NOWRAP
{
        EXCLUDE, "lAccount", "string", "1";
        EXCLUDE, "lAccount0", "string", "2";
        EXCLUDE, "lAmount", "string", "50";
        EXCLUDE, "lTranType", "string", "3";
}
```

Using Real Data Values

When creating interaction diagrams, be sure to include real data values for the arguments in your COM calls rather than the data type names that Rose defaults to in the message signatures. If you fail to supply default data values in the message signatures of your interaction diagrams, your scripts may fail on playback. The interaction diagrams in the sample model provided with Rational PerformanceArchitect provide default data values for your reference. (See *Working with the Sample Model* on page 8.)

NOTE: To display real data values, be sure to configure Rose to display message signatures. To do this, click **Tools** > **Options**, click the **Diagram** tab, and be sure that either **Type Only**, **Name Only**, or **Name and Type** is selected. However, if you change this option, every argument in your diagram will be reset, and you will need to reenter the argument values.

Controlling Object Creation Overhead

Two features of Rational PerformanceArchitect can be especially useful in modeling architectures for certain types of systems, such as an order entry system. These features are the **Automatically include CoInitialize()/CoUninitialize()** option and the script object feature.

In an order entry system, each user generates numerous transactions. This type of application often requires a different architectural approach than an ATM application, in which many users execute a single transaction.

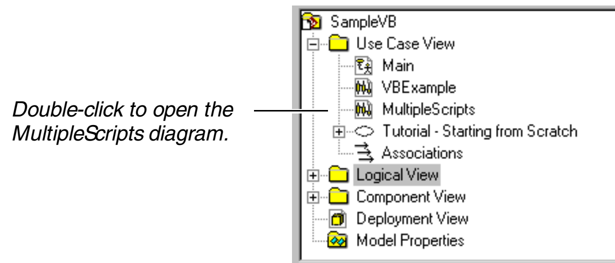
If you select the **Automatically include CoInitialize()/CoUninitialize()** option (**Tools** > **VU Scripting** > **Options**) and generate a script, Rational PerformanceArchitect inserts the `vu_CoInitialize()` and `vu_CoUninitialize()` statements into your script. (These statements are also inserted into the C++ wrappers.) When you run multiple iterations of your script in LoadTest, these statements reset the COM environment and cause any objects created by the script to be recreated with each script iteration. To see how this object creation overhead affects performance, you can compare scripts that include the CoInitialize/CoUninitialize pair with scripts that do not.

Rather than include the `vu_CoInitialize()` and `vu_CoUninitialize()` statements in your script, you can use the script object provided in the Rational Test package to tell Rational PerformanceArchitect to create additional scripts — one to initialize the COM environment and another to uninitialize the COM environment. You can include all three scripts in a single diagram in Rose. Using this technique, you can set up the diagram as follows:

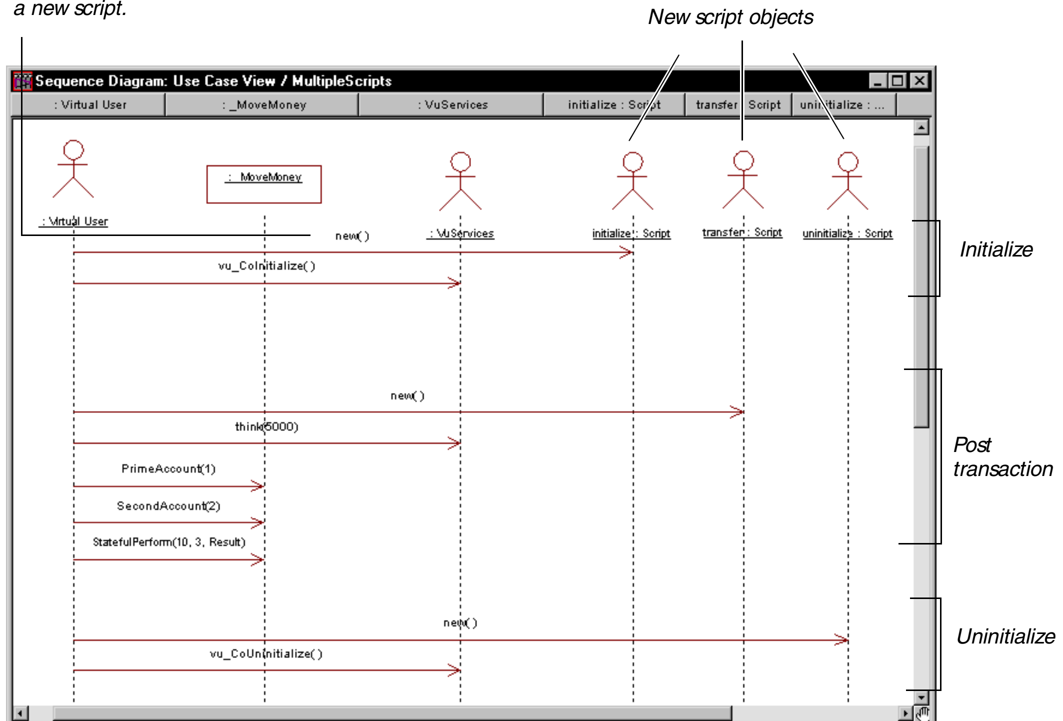
- A `CoInitialize()` script that initializes the COM environment.
- Another script that executes the transaction. If you deselect the **Automatically include CoInitialize()/CoUninitialize()** option, this script will not contain the `vu_CoInitialize()` and `vu_CoUninitialize()` statements, thereby eliminating the overhead of having to recreate the COM object with each iteration.
- A `CoUninitialize()` script that uninitializes the COM environment.

To see how this can be implemented in a Rose diagram:

1. Double-click **MultipleScripts** in the Rose browser for the sample model.



2. View the sequence diagram that is displayed. Note that the diagram includes three script objects and that there is a New() message before each COM call.



To create a scenario such as this in your own diagrams:

1. In Rose, click **Tools** > **VU Scripting** > **Options** and deselect **Automatically include CoInitialize() and CoUninitialize()**.
2. Add the Rational Test package to your script. (See *Adding the RationalTest Package to Your Models* on page 18.)
3. Add a script object for each script that you want to create.

   For example, in the MultipleScripts diagram, there is one script object for transfer, one for initialize, and one for uninitialize.
4. Add a `New()` message before each COM call that you plan to add.

   `New()` tells Rational PerformanceArchitect to start a new script. Each script becomes the receiving object of the `New()` message.
5. After each `New()` message, add a message for each COM call.

   In the sample, these messages are `vu_Coinitialize()`, `PrimeAccount()`, `SecondAccount()`, `StatePerform()` and `vu_CoUninitialize()`.

Sample LoadTest Schedule

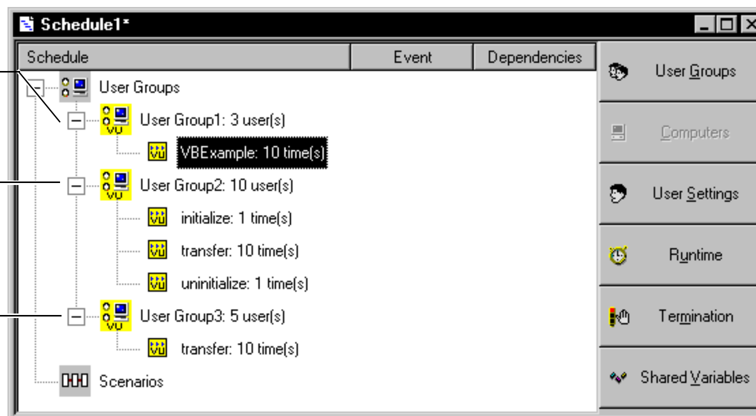
This section shows how your use of the **Automatically include CoInitialize()/ CoUninitialize()** option and the script object can affect a schedule in LoadTest.

In the following sample LoadTest schedule, there are three User Groups — User Group1, User Group2, and UserGroup3 — and four scripts — VBExample, Transfer, Initialize, and Uninitialize.

User Group1 runs the VBExample script 10 times. COM objects are opened and closed with each iteration.

User Group2 runs the Initialize script once, the Transfer script 10 times, and the Uninitialize script once. COM objects are opened and closed once.

User Group3 runs the VBExample script 10 times. The script fails because the COM environment is not initialized.



The following table describes the four scripts:

Script	Description
VBExample	Includes CoInitialize and CoUninitialize COM calls. Can be generated with the Automatically include CoInitialize()/ CoUninitialize() option turned on.
Transfer	Does not includes CoInitialize and CoUninitialize COM calls. Can be generated with the Automatically include CoInitialize()/CoUninitialize() option turned off and with a script object in the Rose diagram.
Initialize	Initializes COM environment. Can be generated with the Automatically include CoInitialize()/CoUninitialize() option turned off and with a script object in the Rose diagram.
Uninitialize	Uninitializes COM environment. Can be generated with the Automatically include CoInitialize()/CoUninitialize() option turned off and with a script object in the Rose diagram.

In this sample, it is expected that User Group2 will outperform User Group1 because of reduced object creation overhead. In addition, the Transfer script will fail to run in User Group3 because of the missing CoInitialize and CoUninitialize calls.

Modeling COM Objects with Rose

COM support is provided in Rational PerformanceArchitect when you import type libraries with the type library import tool in Rose 2000.

The type library import tool defines all the appropriate interfaces, coclasses, and classes and the relationships among them. The structure for each COM object created in the model by the imported type library is the same as the structure created with the ATL (ActiveX Type Library) object creation wizard in Rose 2000.

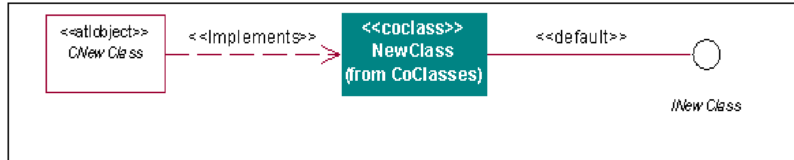
Comparing COM Components in Visual Basis and C++

When you program in Visual Basic, VB masks the complexity of dealing with interfaces and coclasses. You use the class name both to instantiate objects and to call methods on those objects. VB creates a hidden interface, using your class name preceded by a leading underscore character.

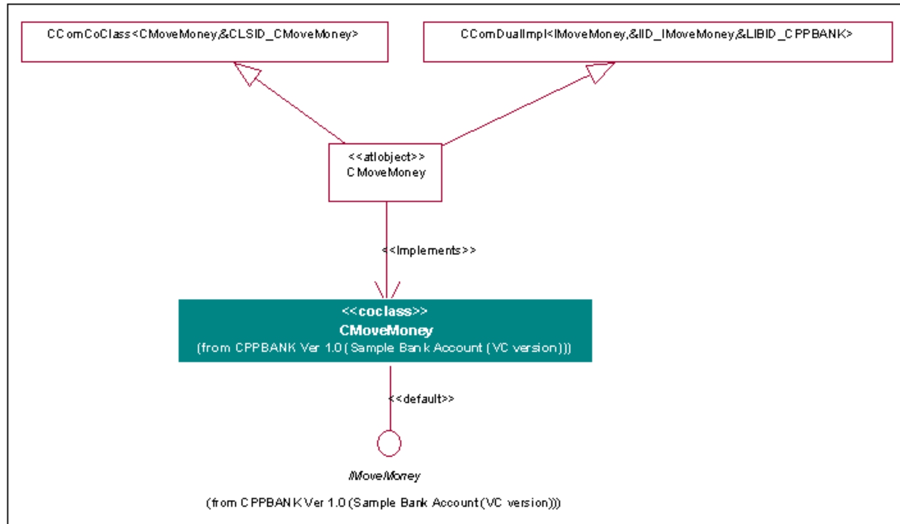
In Visual C++, you typically model interfaces as separate classes and connect them to their coclasses using the UML realizes relationship. In Visual C++, interface names often begin with a leading I and class names often begin with a leading C.

Class Diagram Structure for COM Objects

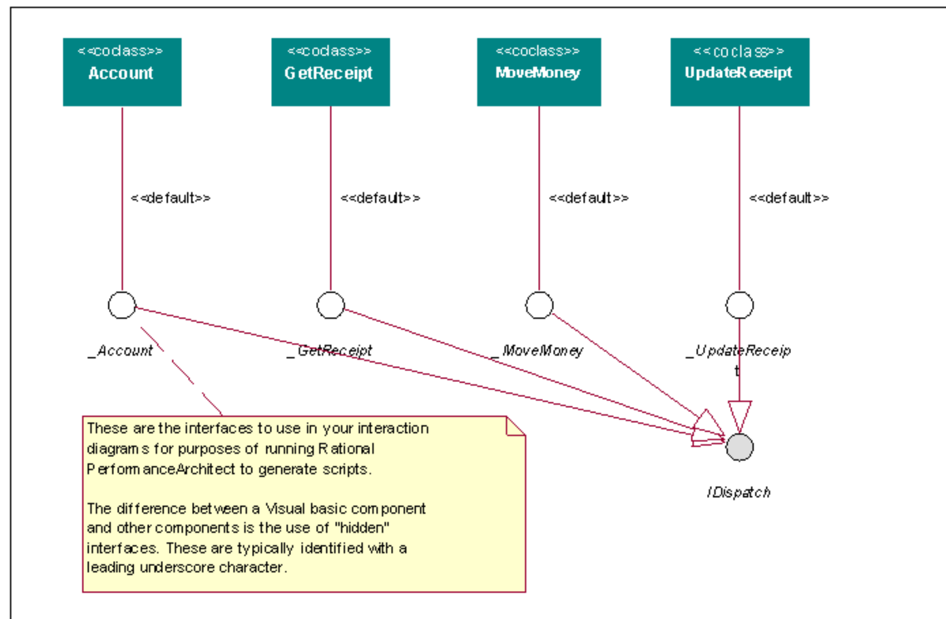
The following figure shows the structure created when you use the ATL object creation wizard in Rose 2000. (You can access this wizard in Rose by clicking Tools > Visual C++ > COM > New ATL Object.) Note that you must implement operations as members of the INewClass interface.



The following figure shows a class diagram for an ATL object imported from a type library for a COM object built using Visual C++:



The following figure shows a class diagram for an ATL object imported from a type library for a COM object built using Visual Basic.



Requirements for COM Objects

Requirements for COM objects in Rational PerformanceArchitect are as follows:

- Rational PerformanceArchitect requires a realize relationship from the coclass to the interface.
- Rational PerformanceArchitect requires that the interface and coclass be realized by a component (in this case Account) in the Component View, as shown in the previous figure. Note that the design of the application will require other classes to be realized by the component.

Qualifying Messages for Inclusion in a Script

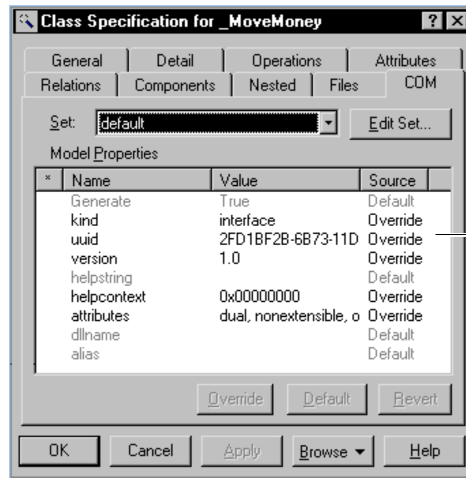
Rational PerformanceArchitect qualifies a message for inclusion in the script as follows:

- The message must be associated with an operation.
- The message must contain a uuid property, which you can find on the COM tab in the specification for the parent class of the message's receiver object.

To verify whether a message qualifies:

1. Right-click the message's receiver object.
2. Click Open Specification.
3. Click Browse and select Browse Class.

   A sample class specification appears as follows:

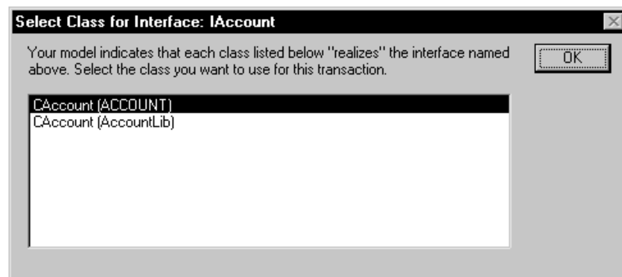


*For a class to qualify as a COM object and be included in a script, the **kind** property must have a value of **interface** and the **uuid** property cannot be blank. If you import the type library using Rose, then these properties will be set properly.*

What Happens When Multiple Classes Realize an Interface

In a model, it is possible to have more than one class of the same name realizing the same interface. When you generate the script, Rational PerformanceArchitect displays a dialog box that prompts you to specify which class you want to use.

In this model, the first CAccount class is part of a package called ACCOUNT, and the second CAccount class is part of a package called AccountLib. Both classes realize the IAccount interface.



How Rational PerformanceArchitect Maps VB Data Types

In the VU language, all data is typed as strings and is passed into the C++ wrapper as a `TCHAR*` data type. Inside the wrapper, these parameters are transformed from a `TCHAR*` data type into local variables before they are passed into the COM object.

When mapping the Visual Basic data types to Visual C++ data types, Rational PerformanceArchitect checks the Type property of the Parameter object. It also inspects the list of properties for each Parameter object to determine whether the Parameter object is passed by value or passed by reference. Specifically, the VC++ type is used for data manipulation inside the wrapper and is passed to the target object by value or by reference according to the dictates of the model.

The following table lists the Visual Basic data types and the corresponding Visual C++ data types used in the wrappers.

When reading the table, you can extrapolate the example to the general case. For example, in the case of Boolean, the variable name `Parm_bool` is the parameter name from the model. The name `wrParm_bool` is the wrapper variable used to pass the transformed data from `Parm_bool` to the object. This pattern is used in all of the examples. Note that two types, Currency and String, use a two-step transformation process.

VB Type	VC++ Type	Example Wrapper Code
Boolean	VARIANT_BOOL	VARIANT_BOOL wrParm_bool = (VARIANT_BOOL)Parm_bool;
Byte	UCHAR	UCHAR wrParm_byte = (UCHAR)Parm_byte;
Currency	CURRENCY	CURRENCY wrParm_curr; wrParm_curr.int64 = _atoi64(Parm_curr);
Date	DATE	DATE wrParm_date = atof(Parm_date);
Double	DOUBLE	DOUBLE wrParm_double = atof(Parm_double);
Integer	SHORT	SHORT wrParm_int = (SHORT)Parm_int;
Long	LONG	LONG wrParm_long = atol(Parm_long);
Object	LPDISPATCH	LPDISPATCH wrParm_obj = (LPDISPATCH)Parm_obj;
Single	FLOAT	FLOAT wrParm_single = (FLOAT)atof(Parm_single);
String	BSTR	BSTR BSTRResult;
Variant	VARIANT	VARIANT wrParm_byval = _variant_t(Parm_byval);

Sample VU Script

```
#include <VU.h>
external_C string strResult;        /*for string results*/
external_C int intResult;          /*for non-string results*/
external_C string pszLogPass;      /*log message for success*/
external_C string pszLogFail;      /*log message for fail*/
external_C int func vu_CoInitialize() {}  /*initialize COM*/
external_C int func vu_CoUninitialize() {} /*unitialize COM*/
external_C int func MoveMoney_PrimeAccount(lAccount)
string lAccount,;
{}
external_C int func MoveMoney_SecondAccount(lAccount)
string lAccount,;
{}
external_C int func MoveMoney_StatefulPerform(lAmount,lTranType)
string lAmount,lTranType,;
{}
/* Main */
{
        int iRetval = 0;
        /* ************************************************************** */
        /* The following code is common initialization code for all scripts */
        /* ************************************************************** */
        push Timeout_scale = 200; /* Set timeouts To 200% of maximum response Time *
        push [Think_avg = 0, Think_dist = "NEGEXP", Think_def = "LS"];
        push Timeout_val = 120000;/* Set minimum Timeout_val to 2 minutes          *
        /* ************************************************************** */

        vu_CoInitialize();  /* Automatically generated - vu_CoInitialize() */
        VBExample = datapool_open("VBExample");
        datapool_fetch(VBExample);
        /* ******************************************************************** */
        /* Operations mapped in the sequence diagram                            */
        /* ******************************************************************** */
        /* think() message from sequence diagram */
        set Think_avg = 2500;
        emulate ["MoveMoney_PrimeAccount"] iRetval = MoveMoney_PrimeAccount(
                        datapool_value(VBExample, "lAccount")),
                        pszLogPass, pszLogFail;
        emulate ["MoveMoney_SecondAccount"] iRetval = MoveMoney_SecondAccount(
                        datapool_value(VBExample, "lAccount0")),
                        pszLogPass, pszLogFail;
        emulate ["MoveMoney_StatefulPerform"] iRetval = MoveMoney_StatefulPerform(
                        datapool_value(VBExample, "lAmount"),
                        datapool_value(VBExample, "lTranType")),
                        pszLogPass, pszLogFail;

        /* ******************************************************************** */
        vu_CoUninitialize(); /* Automatically generated - CoUninitialize() */
        pop [Think_avg, Think_dist, Think_def, Timeout_val, Timeout_scale];
}
DATAPOOL_CONFIG "VBExample" OVERRIDE DP_SEQUENTIAL DP_SHARED DP_NOWRAP
{
        EXCLUDE, "lAccount", "string", "1";
        EXCLUDE, "lAccount0", "string", "2";
        EXCLUDE, "lAmount", "string", "50";
        EXCLUDE, "lTranType", "string", "3";
}
```

Sample C++ Wrapper Generated for the VBExample Script

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <comdef.h>
#include <tchar.h>

#ifndef _WIN32_DCOM
#define _WIN32_DCOM
#endif
#define DLLEXPORT __declspec(dllexport)

#import "D:\Program Files\Rational\Rose 2000\RPA\Sample.vb\RpaDemoVB.dll"

using namespace RpaDemoVB;

TCHAR sDat1[256]; /*buffer for strResult*/
TCHAR sDat2[256]; /*buffer for pszLogPass*/
TCHAR sDat3[256]; /*buffer for pszLogFail*/
extern "C" {
DLLEXPORT int     intResult;
DLLEXPORT TCHAR*  strResult = sDat1;
DLLEXPORT int*    addr_intResult(void) {return &intResult;}
DLLEXPORT TCHAR** addr_strResult(void) {return &strResult;}
DLLEXPORT TCHAR*  pszLogPass = sDat2;
DLLEXPORT TCHAR*  pszLogFail = sDat3;
DLLEXPORT TCHAR** addr_pszLogPass(void) {return &pszLogPass;}
DLLEXPORT TCHAR** addr_pszLogFail(void) {return &pszLogFail;}

DLLEXPORT _MoveMoney* p_MoveMoney = NULL;
DLLEXPORT LONG* addr_p_MoveMoney(void)   {return (LONG*)&p_MoveMoney;}

DLLEXPORT int MoveMoney_PrimeAccount(TCHAR* lAccount)
{
    LONG rc = 1;
    _tcscpy(pszLogPass, "Pass: ");
    _tcscpy(pszLogFail, "Fail: ");

    HRESULT HRESULTResult;
        //Declare and initialize variables for parameters
        //to be passed.
    LONG wrlAccount = atol(lAccount);

    _MoveMoneyPtr p_MoveMoneyPtr = NULL;
    try {

        // Use an existing interface reference if available.
        // To create the object on every iteration, model the
        // vu_CoInitialize and vu_CoUninitialize operations in
        // your sequence diagram.

        if (p_MoveMoney == NULL) {
            p_MoveMoneyPtr = _MoveMoneyPtr(__uuidof(MoveMoney));

        // Save raw interface ptr in process global memory.
        // Increment the reference count so the smart pointer
        // does not clean up when we lose scope returning control
        // to the script.  The vu_CoUninitialize() function
        // releases this reference.

            p_MoveMoney = p_MoveMoneyPtr.GetInterfacePtr();
            p_MoveMoney->AddRef();
        }
```

```
        else {
            p_MoveMoneyPtr = _MoveMoneyPtr(p_MoveMoney);
        }

        //Call the COM interface using a smart pointer.
        //But, if the object platform is Visual basic then
        //use the raw interface pointer.
        HRESULT hr = p_MoveMoney->put_PrimeAccount(wrlAccount,&HRESULTResult);

        if (FAILED(hr))
            _com_raise_error(hr);

    }
    catch (_com_error e) {
        rc = 0;    // Return 0 to VU to signal a failure.
        _tcscat(pszLogFail, (TCHAR*)e.ErrorMessage());  //com_error message to VU log
    }
    catch (...) {
        rc = 0;
        _tcscat(pszLogFail, "Failure not COM-related.");
    }
    return rc;
}

DLLEXPORT int MoveMoney_SecondAccount(TCHAR* lAccount)
{
    LONG rc = 1;
    _tcscpy(pszLogPass, "Pass: ");
    _tcscpy(pszLogFail, "Fail: ");

    HRESULT HRESULTResult;
        //Declare and initialize variables for parameters
        //to be passed.
    LONG wrlAccount = atol(lAccount);

    _MoveMoneyPtr p_MoveMoneyPtr = NULL;
    try {

        // Use an existing interface reference if available.
        // To create the object on every iteration, model the
        // vu_CoInitialize and vu_CoUninitialize operations in
        // your sequence diagram.

        if (p_MoveMoney == NULL) {
            p_MoveMoneyPtr = _MoveMoneyPtr(__uuidof(MoveMoney));

        // Save raw interface ptr in process global memory.
        // Increment the reference count so the smart pointer
        // does not clean up when we lose scope returning control
        // to the script.  The vu_CoUninitialize() function
        // releases this reference.

            p_MoveMoney = p_MoveMoneyPtr.GetInterfacePtr();
            p_MoveMoney->AddRef();
        }
        else {
            p_MoveMoneyPtr = _MoveMoneyPtr(p_MoveMoney);
        }

        //Call the COM interface using a smart pointer.
        //But, if the object platform is Visual basic then
        //use the raw interface pointer.
        HRESULT hr = p_MoveMoney->put_SecondAccount(wrlAccount,&HRESULTResult);

        if (FAILED(hr))
```

```
            _com_raise_error(hr);

    }
    catch (_com_error e) {
        rc = 0;    // Return 0 to VU to signal a failure.
        _tcscat(pszLogFail, (TCHAR*)e.ErrorMessage());  //com_error message to VU log
    }
    catch (...) {
        rc = 0;
        _tcscat(pszLogFail, "Failure not COM-related.");
    }
    return rc;
}

DLLEXPORT int MoveMoney_StatefulPerform(TCHAR* lAmount, TCHAR* lTranType)
{
    LONG rc = 1;
    _tcscpy(pszLogPass, "Pass: ");
    _tcscpy(pszLogFail, "Fail: ");

    BSTR* BSTRResult;
        //Declare and initialize variables for parameters
        //to be passed.
    LONG wrlAmount = atol(lAmount);
    LONG wrlTranType = atol(lTranType);


    _MoveMoneyPtr p_MoveMoneyPtr = NULL;
    try {

        // Use an existing interface reference if available.
        // To create the object on every iteration, model the
        // vu_CoInitialize and vu_CoUninitialize operations in
        // your sequence diagram.

        if (p_MoveMoney == NULL) {
            p_MoveMoneyPtr = _MoveMoneyPtr(__uuidof(MoveMoney));

        // Save raw interface ptr in process global memory.
        // Increment the reference count so the smart pointer
        // does not clean up when we lose scope returning control
        // to the script.  The vu_CoUninitialize() function
        // releases this reference.

            p_MoveMoney = p_MoveMoneyPtr.GetInterfacePtr();
            p_MoveMoney->AddRef();
        }
        else {
            p_MoveMoneyPtr = _MoveMoneyPtr(p_MoveMoney);
        }

        //Call the COM interface using a smart pointer.
        //But, if the object platform is Visual basic then
        //use the raw interface pointer.
        HRESULT hr = p_MoveMoney->StatefulPerform(wrlAmount,wrlTranType,&BSTRResult);

        if (FAILED(hr))
            _com_raise_error(hr);

        _tcscat(pszLogPass, (TCHAR*)_bstr_t(BSTRResult));
        _tcscpy(strResult, (TCHAR*)_bstr_t(BSTRResult));
    }
    catch (_com_error e) {
        rc = 0;    // Return 0 to VU to signal a failure.
        _tcscat(pszLogFail, (TCHAR*)e.ErrorMessage());  //com_error message to VU log
```

```
    }
    catch (...) {
        rc = 0;
        _tcscat(pszLogFail, "Failure not COM-related.");
    }
    return rc;
}

DLLEXPORT int vu_CoInitialize()
{
    ::CoInitialize(NULL);
    return 1;
}

DLLEXPORT int vu_CoUninitialize()
{
    //Release the raw interface pointer and set it to 0
    if (p_MoveMoney) {
    try {

        p_MoveMoney->Release();
        p_MoveMoney = 0;

        }

        catch (...) {
        //Catch everything so we do not crash.
        }
    }
    //Release the raw interface pointer and set it to 0
    if (p_MoveMoney) {
    try {

        p_MoveMoney->Release();
        p_MoveMoney = 0;

        }

        catch (...) {
        //Catch everything so we do not crash.
        }
    }
    //Release the raw interface pointer and set it to 0
    if (p_MoveMoney) {
    try {

        p_MoveMoney->Release();
        p_MoveMoney = 0;

        }

        catch (...) {
        //Catch everything so we do not crash.
        }
    }
    ::CoUninitialize();
    return 1;
} }
```

Troubleshooting

This section provides troubleshooting techniques you can use when running Rational PerformanceArchitect.

If	Then
Wrappers fail to build and you see a message such as `Program is not an executable file`.	See *Wrappers Fail to Build* on page 35.
Wrappers fail to build and the build results are not displayed	See *Configuring Notepad to Always Display Build Errors* on page 36.
A `COM not initialized error` appears when you run a LoadTest schedule.	You failed to call vu_CoInitialize and vu_CoUninitialize in your script. To correct this error, select the **Automatically include CoInitialize and CoUninitialize** option in the Rational PerformanceArchitect Options dialog box and regenerate the script; or, model the call in your diagram and regenerate the script. For more information, see *Controlling Object Creation Overhead* on page 22.
Script fails to open in Robot after script has been generated, even with the Launch Robot option selected.	Be sure to exit out of Robot before generating a script.

Wrappers Fail to Build

Wrapper compilation may fail for a variety of reasons related to the data types of parameters. For example, you may have uncovered a data type conversion that Rational PerformanceArchitect does not handle properly, or you may have exposed an issue in your model.

In addition, it is possible that Notepad may fail to initialize and display the results of a wrapper build when build errors occur.

Problems with Visual Basic Applications

For Visual Basic components, you must verify that you have properly modeled the `ByVal` and `ByRef` properties using `ByVal` in the parameter name or using the Model Assistant. For information about the Model Assistant, see the *Using Rose Visual Basic* manual.

In addition, this version of Rational PerformanceArchitect generates interface wrappers that use early binding. Visual Basic applications may use late binding depending on the data types for functions and parameters that are being used.

Missing Libraries

Wrappers may also fail to build because of various undefined COM operations. Rational PerformanceArchitect generates code for `#import` statements in the VC++ interface wrapper code. One `#import` statement is generated for each COM object included in a given diagram. In some cases, your transaction may depend on type libraries that are not directly referenced in your sequence diagram. In this case, you need to determine which library is missing and add a `#import` statement for it manually.

Exceeding the Maximum Wrapper Size

Wrappers will fail to build if the source file for the wrapper is larger than 64k bytes.

Program is Not an Executable File Message

If the wrappers fail to build and you see a message such as `Program is not an executable file`:

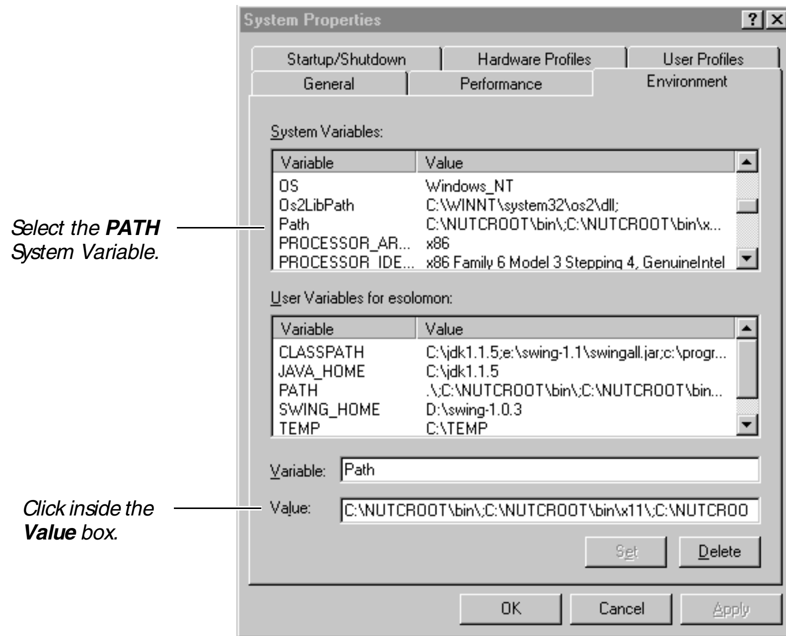
1. Verify that you have the C++ compiler installed on your computer.
2. Click **Automatically build COM interface wrappers** in the Rational PerformanceArchitect Options dialog box.
3. Regenerate the script.
4. If step 3 fails, run the Vsvars32 batch file, located in the Visual C++ bin directory.
5. Then, run the rpa.bat file. (See *Generating the Script and Wrappers in Rose* on page 13.)

Configuring Notepad to Always Display Build Errors

On Windows NT computers, Rational Suite PerformanceStudio may prevent Windows from locating the Notepad executable and cause Windows to fail to display the results of a wrapper build when build errors occur.

To ensure that Notepad will display build errors:

1. Click **Start > Settings > Control Panel > System** and click the **Environment** tab.



2. Select the PATH System Variable.
3. Click inside the Value box at the bottom of the dialog (without making any changes).
4. Click Set and click OK.
5. Regenerate the script and wrappers by running the rpa.bat command file associated with the wrapper or by clicking Tools > VU Scripting > Generate Script in Rose.

   Be sure that the Automatic build COM Interface Wrappers option is selected when you generate the script.

Asynchronous Environments

Rational PerformanceArchitect generates scripts for testing the performance of COM/DCOM applications in a synchronous client/server environment. Visual Basic client applications that respond to asynchronous server-initiated requests (callbacks) using VB events are not supported by Rational PerformanceArchitect.

Runtime Registry Settings

Rational PerformanceArchitect maintains several runtime registry settings in HKEY_CURRENT_USER\Software\BasicScript Program Settings\Rational PerformanceArchitect. The following table describes the keys and their legal values:

Key	Value
`AutoBuild`	`TRUE`. Build wrappers when generating script. `FALSE`. Do not build wrappers.
`CompilerLocation`	Fully-qualified location and name of vcvars32.bat.
`InitializeCOM`	`TRUE`. Include vu_CoInitialize() and vu_CoUninitialize calls. `FALSE`. Do not include these calls. User must model them explicitly.
`LaunchRobot`	`TRUE`. Launch Robot when script generation completes. `FALSE`. Do not launch Robot.
`TypelibHistory`	Collection of key values for the list of previously used type libraries.
`UseDefaultSettings`	`TRUE`. Set all options to their default values the next time VU Scripting is called. `FALSE`. Use the settings as specified by the user. **Note**: This is a system setting used during product installation.
`UseDiagramList`	`TRUE`. Display a list of diagrams in the model. `FALSE`. Use the current diagram.

Index

A

ActiveX Type Library. See also ATL
ATL object creation wizard 25, 26

B

binding, early vs. late 36

C

Class Specification dialog box 7
coclasses with Visual Basic 25
CoInitialize/Uninitialize runtime functions 14
collaboration diagrams 5
COM objects
  interfaces imported from Visual Basic 8
  requirements for 27
component diagrams 8

D

data types, mapping VB to VC++ 29
datapools
  DATAPOOL_CONFIG section of script 19
  definition 8
  support for in RationalTest package 5

E

early binding 36
emulate command 7, 18

F

fat client 1
FileName property, in Component Specification 7

I

INFO SERVER command 5
installation instructions 4
interaction diagrams 1, 5
interfaces
  COM 8
  leading underscore character in name 25
  realize relationship from coclass 27

K

kind property 28

L

late binding 36
LoadTest 3

M

MultipleScripts sequence diagram 23

N

Notepad, configuring to display build errors 36

O

OLE objects. *See also* stdole
operation signatures, in Rose 22
options, VU Scripting 13

P

performance tests, running 16

R

Rational Test package 3, 18
realize relationship 27
receiver objects 6
registry settings 38
repository 15
requirements, PerformanceArchitect 3, 4
round-trip engineering 8, 11

S

schedules, in Rational LoadTest 24
scripts
- generating 13
- support for generating multiple 5

sequence diagrams 5
set Think_Avg command 18
StatefulPerform message 12
stdole 8, 11

T

thin client 1
think time 18
Timer actor 5

U

underscore characters, in COM interfaces 8
user groups, in Rational LoadTest 24
uuid property 27, 28

V

virtual user actor 5, 11
virtual user scripts, definition 1
VU actor 5
VU Scripting options 13
Vu_CoInitialize and Vu_CoUninitialize statements 14, 22
VuServices actor 5, 11

W

wrappers
- FileName property in 7
- generating 13
- troubleshooting 35