

Using Rational QualityArchitect

VERSION 2001A.04.00

PART NUMBER 800-024534-000

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

Rational[®]
the **e-development** company™

IMPORTANT NOTICE

COPYRIGHT

Copyright ©2001, Rational Software Corporation. All rights reserved.

Part Number: 800-024534-000

PERMITTED USAGE

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION ("RATIONAL") AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

TRADEMARKS

Rational, Rational Software Corporation, the Rational logo, Rational the e-development company, ClearCase, ClearQuest, Object Testing, Object-Oriented Recording, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Purify'd, Quantify, Rational Apex, Rational CRC, Rational PerformanceArchitect, Rational Rose, Rational Suite, Rational Summit, Rational Unified Process, Rational Visual Test, Requisite, RequisitePro, SiteCheck, SoDA, TestFactory, TestMate, TestStudio, and The Rational Watch 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.

Microsoft, the Microsoft logo, the Microsoft Internet Explorer logo, DeveloperStudio, Visual C++ , Visual Basic, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

FLEXIm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

PATENT

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

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 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

WARRANTY DISCLAIMER

This document and its associated software may be used as stated in the underlying license agreement. Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.

Contents

Preface	ix
Audience	ix
Other Resources	ix
Contacting Rational Technical Publications	x
Contacting Rational Technical Support	x
1 Introduction	1
What You Need to Get Started	1
Supported Environments	2
If You Are Testing Enterprise JavaBeans	2
If You Are Testing COM, DCOM, and COM+ Components	2
Installing Rational QualityArchitect	2
Quick Start	2
Step 1: Working in the Rational Administrator	2
Step 2: Working in Rose	3
Step 3: Working in Your IDE	3
Working in VisualAge for Java (VAJ)	3
Working in Visual Cafe	4
Working In Visual Basic	4
2 Basic Concepts	5
Test Script Basics	5
Test Types	5
Storing Your Tests Scripts	6
Templates	7
Scenario Tests	8
Support for Performance Testing	10
Support for Verification Points	10
Templates for Scenario Test Generation	11

Message Signatures and Data Correlation	11
Signature Options	12
Message Syntax	13
Editing Message Signatures	14
How Message Signatures Correlate to Datapool Fields	14
Message Signature Examples	15
Stubs	16
Templates for Stub Generation	19
EJB	19
COM/VB	20
Test Script Services	21
Datapools	21
Datapools in Scenario Tests	21
Datapools in Unit Tests	22
Data Types	23
Verification Points	23
How Data Is Verified	23
Static, Dynamic, and Manual Verification Points	24
Database Verification Point	25
The Query Builder	26
Extensibility	26
3 Testing Enterprise JavaBeans	27
Overview	27
EJB TestScripts	28
Package Requirements	28
Requirements for EJB Testing	29
Supported Environments	29
Rational TestManager Playback Requirements	31
Visual Age/WebSphere Requirements	31
Visual Cafe/WebLogic Requirements	31
Visual Cafe/WebSphere Requirements	32
EJB Session Recorder Requirements	32
The Rational Bank Account Sample Application	32
Setting Up the Sample Application for VAJ/WebSphere	33
Importing the Sample into VAJ	33
Creating the Database	34
Configuring VAJ to Use the New Sample Database	35
Adding EJB Support	35
Deploying the Sample Application in the VAJ Test Environment	36
Importing Swing and the Rational Support Classes	39

Deploying the Sample Application in WebLogic	40
Deploying the Sample Application to the Sun J2EE Reference Server	42
Configuring Visual Cafe to Run the Sample Application	43
Adding Account Information to the Rational Bank Account Sample	44
Reverse Engineering a Deployed EJB into Rose	45
Generating Test Assets	45
Generating EJB Test Scripts	46
Generating Unit Test Scripts from a Rose Model	46
Generating Test Scripts with the EJB Session Recorder	48
Creating a Datapool for the Unit Test	49
Generating Stubs for the Unit Test	51
Creating Lookup Tables	52
Importing Test Assets into VAJ	52
Importing Unit Tests Into VAJ	53
Importing Stubs into VAJ	53
Executing Test Scripts	55
Executing Test Scripts from VAJ	55
Executing Test Scripts from Visual Cafe	56
Executing Test Scripts from TestManager	56
Using EJB Scenario Tests to Test Transactions	57
Using the Java Query Builder to Add Verification Points	59
Connecting to the Database via JDBC	60
JDBC Driver	60
JDBC URL	61
User Name	61
Password	61
Designing a Custom Query Statement	62
Using the Query Design Wizard	62
Viewing and Verifying the SQL Query Results	65
Using the EJB Session Recorder	66
EJB Session Recorder Requirements	67
Setting Up the Classpath for the EJB Session Recorder	68
Setting Up the Java Runtime Environment (JRE)	69
Starting the EJB Session Recorder	69
Starting the EJB Session Recorder from the Console	69
Starting the EJB Session Recorder from Visual Cafe	70
Starting the EJB Session Recorder from Visual Age for Java 3.0x	71
Starting the EJB Session Recorder from Visual Age for Java 3.5	74

Using the EJB Session Recorder with the Sample Application	75
Starting a Recording Session	76
Connecting to an EJB	76
Interacting with the Home Interface	79
Interacting with the Remote Interface	80
Inserting a Verification Point	85
Viewing the XML Log	86
Generating a Test Script from the XML Log	86
4 Testing COM Components.	87
Overview	87
Testing Existing Objects	87
Testing with Iterative Development	88
Programming in Visual Basic	88
Requirements for Testing COM Components	88
Working with the Sample Model.	89
Understanding the Component View	89
Understanding the Logical View	90
Generating Test Assets	90
Generating Unit Test Scripts from a Rose Model	90
Creating a Datapool	93
Generating Stubs for the Unit Test	94
Creating Lookup Tables	95
Executing Test Scripts	96
Using COM Scenario Tests to Test Transactions	96
Using the OLE DB Query Builder to Add Verification Points	99
Connecting to the Database via OLE DB	100
Designing a Custom SQL Statement	102
Using the Query Design Wizard	102
Reviewing the Record Set	105
Accepting the Query	105
DatabaseVP Advanced Options	106
A Template Replacement Variables	107
Replacement Variables for Unit Test Generation Templates	107
Variables Used in All Unit Test Generation Templates	108
Variables Used Only with the COM/Visual Basic Templates	111
Variables Used Only with the EJB Templates.	112

Replacement Variables for Scenario Test Generation Templates	114
Variables Used in Both the COM/VB and EJB Templates	115
Variables Used Only in the COM/VB Scenario Test Templates	117
Variables Used Only in the EJB Scenario Test Templates	118
Replacement Variables for Stub Generation Templates.	119
Replacement Variables for the COM/VB Stub Templates	119
Replacement Variables for the EJB Stub Templates	121
B Troubleshooting	125
Resource File Not Found	125
Symptom	125
Explanation	125
EJB Class or Interface Not Found Messages	125
Glossary.	127
Index.	131

Preface

This manual provides conceptual information and task-oriented guidelines for using Rational QualityArchitect. QualityArchitect is a collection of integrated tools for testing middleware components built with technologies such as Enterprise JavaBeans and COM.

Audience

This guide is intended for all members of the development team who design, write, or edit test scripts to be used for testing Enterprise JavaBeans and COM components. A solid foundation in the target test script language is assumed.

Other Resources

This guide is available as a printed manual and in electronic form as HTML and PDF files.

To access the HTML version:

- Start Rose and click **Tools > QualityArchitect > Online Manual**.

The PDF version of this manual is available on the *Rational Solutions for Windows* Online Documentation CD.

Context-sensitive Help is available for QualityArchitect from within Rose.

To access the online Help:

- 1 Click **Tools > QualityArchitect > Toolbar**.
- 2 When the toolbar is displayed, press **F1**.

The Help will appear after several seconds.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

Contacting Rational Technical Support

If you have questions about installing, using, or maintaining this product, contact Rational Technical Support as follows:

Your Location	Telephone	Facsimile	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4545-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

When you contact Rational Technical Support, please be prepared to supply the following information:

- Your name, telephone number, and company name
- Your computer's make and model
- Your operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously-reported problem)

Rational QualityArchitect is a powerful collection of integrated tools for testing middleware components built with technologies such as Enterprise JavaBeans (EJB) and COM.

QualityArchitect, in conjunction with Rational Rose, generates test scripts for components and interactions in your Rose model. Once generated, the test scripts can be edited and run from your development environment or from Rational TestManager.

With QualityArchitect, you can:

- Generate test scripts that unit test individual methods or functions in a component-under-test.
- Generate test scripts that drive the business logic in a set of integrated components. Scripts can be generated directly from Rose interaction diagrams or, in the case of EJBs, from live components using the EJB Session Recorder.
- Generate stubs that allow you to test components in isolation, apart from other components called by the component-under-test.
- Track code coverage through Rational PureCoverage and model-level coverage through Rational TestManager.

What You Need to Get Started

To develop tests with QualityArchitect, you need:

- A workstation running Microsoft NT 4.0, Windows 2000, Windows 98, or Windows ME.
- Rational Rose
- A project for storing your test assets (created with Rational Administrator)

Supported Environments

QualityArchitect supports the environments listed in the following sections:

If You Are Testing Enterprise JavaBeans

- Version 1.1.x and later of the Sun Java Developer Kit (JDK)
- Version 3.02 or later of IBM Visual Age for Java, or Version 3.1 or later of Visual Cafe, Enterprise Edition
- Version 3.02 or later of IBM WebSphere, Advanced Edition, or Version 4.5.2 or later of BEA WebLogic

If You Are Testing COM, DCOM, and COM+ Components

QualityArchitect can test components in any language that realizes COM interfaces. Microsoft Visual Basic 6.0 is required to compile and run test scripts.

Installing Rational QualityArchitect

Rational QualityArchitect can be installed as part of Rational Suite DevelopmentStudio, Rational Suite Enterprise, or Rational Rose Enterprise. For information about the Rational Suite installations, see the *Installing Rational Suite* manual. For information about the Rose installation, see the *Installing Rational Rose* manual.

The installation process adds a QualityArchitect subdirectory to your Rational Test directory.

Quick Start

Use the procedures in this section to get started with Rational QualityArchitect.

Step 1: Working in the Rational Administrator

Before you generate test scripts, use the Rational Administrator to create a project. Each project contains a datastore for storing test assets, such as datapools, lookup tables, and log files.

For details, see the *Using the Rational Administrator* manual or the Administrator Help.

Step 2: Working in Rose

- 1 If you have an existing component, reverse engineer it into your Rose model. (Skip this step if you are already modeling your system and components in Rose.)

For EJBs, a good way to do this is with the Rose/Java Add-in. Simply make a jar file and drop the jar file on a class diagram. Be sure to include only the source (.java) files for the Remote interface, the Home interface, and the Bean itself. Do not import any .class files compiled from the source files because these will not be properly reverse-engineered with parameter names.

For more information, see *Reverse Engineering Java Source* in your Rose J online Help.

- 2 Choose a template for the test generation process. This template contains replacement variables that get populated when you generate the test script. (A template will be provided for you by default.) For more information, see *Templates* on page 7 and *Template Replacement Variables* on page 107.

In Rose, right-click a model element to test and click **QualityArchitect > Select Unit Test Template**.

- 3 To create a unit test, right-click the item to test and click **QualityArchitect > Generate Unit Test**.

For information about testing your business logic, see *Using EJB Scenario Tests to Test Transactions* on page 57 or *Using COM Scenario Tests to Test Transactions* on page 96. For information about generating EJB test scripts with the Session Recorder, see *Using the EJB Session Recorder* on page 66.

- 4 If this is your first time generating a test script, you will be prompted to select a source script directory in which to store your scripts. For more information, see *Storing Your Tests Scripts* on page 6.

Step 3: Working in Your IDE

After you generate your test assets, you must add them to a project in your IDE, where you can run the tests.

This section includes the basic steps for working in your IDE.

Working in VisualAge for Java (VAJ)

If you are using IBM Visual Age for Java (VAJ), you need to import test assets into the VAJ repository. For details, see *Importing Test Assets into VAJ* on page 52.

Working in Visual Cafe

In Visual Cafe:

- 1 Open a project.
- 2 Add the test scripts to the project. See *Adding an existing file to a project* in the Visual Cafe Help.
- 3 Update the CLASSPATH in Visual Cafe, adding references to `rttseajava.jar`, `rttssjava.jar`, and `rational_ct.jar`. (See *Requirements for EJB Testing* on page 29.)
- 4 Start the WebLogic server.
- 5 Click **Project > Execute** to run the test.

Working In Visual Basic

Every generated test of a COM component results in the creation of a Visual Basic project and several other files. For a list and description of these generated files, see *Programming in Visual Basic* on page 88.

To run your test in Visual Basic:

- 1 Open the Visual Basic project file.
- 2 Edit the `.cls` file as needed.
- 3 Click **Run > Start with Full Compile**.

This chapter discusses several basic concepts that will help you take advantage of the features provided with Rational QualityArchitect. Topics include:

- Test script basics
- Templates
- Scenario tests
- Stubs
- Test script services

For information about concepts that apply to specific component models, see the chapters that describe each particular model.

Test Script Basics

QualityArchitect generates *test scripts* that drive and validate the component-under-test. Test scripts are generated in various languages, depending on the type of component you are testing. The following table lists the test script languages for each component type:

Component Type	Language
Enterprise JavaBeans	Java
COM components	Visual Basic

Test Types

There are two types of test scripts that QualityArchitect can generate from Rose:

- Unit tests
- Scenario tests

A *unit test* tests the behavior of an individual method or operation.

A *scenario test* tests the behavior of components as specified in an interaction diagram. These tests are intended to replicate the sequence of events in a transaction, and as such, test the implementation of the transaction. You can use interaction diagrams to construct simple scenario tests involving a single component or complex tests involving multiple components.

In addition, there is another kind of test script that you can generate with the EJB Session Recorder. For details, see *Using the EJB Session Recorder* on page 66.

Storing Your Tests Scripts

Projects for storing test assets are created in the Rational Administrator. Each project can contain several datastores—for example, a datastore for test assets, a datastore for requirements, and a datastore for change requests.

Certain test assets for QualityArchitect, such as datapools, lookup tables, and log files are stored in the test datastore. Test scripts, however, can be stored independently of the test datastore, in a location of your choice, where they can be placed under source control. When you generate your first test script, you will be prompted to log in to a project and to select a directory for storing your test scripts. QualityArchitect maintains an association between this directory and the project's test datastore.

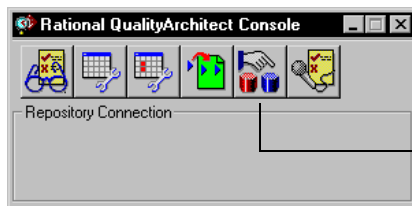
Note: If you want to share your test scripts with a team, be sure to use a UNC path (`\\server-name\directory-path`) to create the project and a UNC path to specify your script source directory. Failure to do both of these will result in a project and script source that cannot be used in a group environment.

To create a new project with the Rational Administrator:

- 1 Click **Start > Programs <Rational ...> Rational Test > Rational QualityArchitect Console**.

where *<Rational ...>* is the name of the Rational product you have installed—for example, Rational Suite DevelopmentStudio.

- 2 When the Rational QualityArchitect console appears, click the **Administrator** icon on the console.



Click to open the Administrator.

3 Click **File > New Project**.

For more details, see the *Using the Rational Administrator* manual.

Templates

QualityArchitect uses templates to provide structure and common code to generated scripts and stubs. *Templates* are simply ASCII text files with replacement variables. The code generators supplied with QualityArchitect replace the variables in the templates with real code derived from Rose model elements.

All of the Rational templates can be customized. To create your own template, simply copy one of the supplied templates and edit it as needed. For example, to provide more logging information in your scripts, copy one of the supplied templates, add logging code, and generate tests with the new template.

Templates also provide a means for supporting additional environments, such as application servers. Generally, to add support for a new environment, you adjust a standard template, primarily in the initialization code, and then generate your test scripts using the new templates.

Rational has included several templates for your use. These templates can be logically grouped into the following general categories:

- Templates for unit test generation
- Templates for scenario test generation
- Templates for stub generation

Within each general category there are several templates for the various component models. You can browse through the Rational Test\QualityArchitect\Template directory to view the list of templates that are installed.

The following table lists the templates QualityArchitect uses for unit test generation:

Template	Description
weblogic_home.template	Template for testing the methods in the home interface of EJBs on a WebLogic server. Use this template: <ul style="list-style-type: none">▪ To ensure that the remote interface can be created successfully▪ As a check before writing or running method-level tests
weblogic_remote.template	Template for testing the methods in the remote interface of EJBs on a WebLogic server.

Template	Description
websphere_home.template	Template for testing the methods in the home interface of EJBs in the WebSphere environment.
websphere_remote.template	Template for testing the methods in the remote interface of EJBs in the WebSphere environment.
sunj2ee_home.template	Template for testing the methods in the home interface of EJBs in the Sun J2EE environment.
sunj2ee_remote.template	Template for testing the methods in the remote interface of EJBs in the Sun J2EE environment.
TestNameScript.cls	Template for the Visual Basic .cls file that is generated to test a COM component. The .cls file is the template for the actual test program.
TestName.vbp	Template for the Visual Basic project file that is generated to test a COM component.
TestNameMain.bas	Template for the main program that calls the test program in the .cls file. Visual Basic requires a <i>main</i> program to begin execution.
TestName.res	Template for the resource file that is created to support datapools.

For more information about templates, see *Templates for Scenario Test Generation* on page 11, *Templates for Stub Generation* on page 19, and *Template Replacement Variables* on page 107.

Scenario Tests

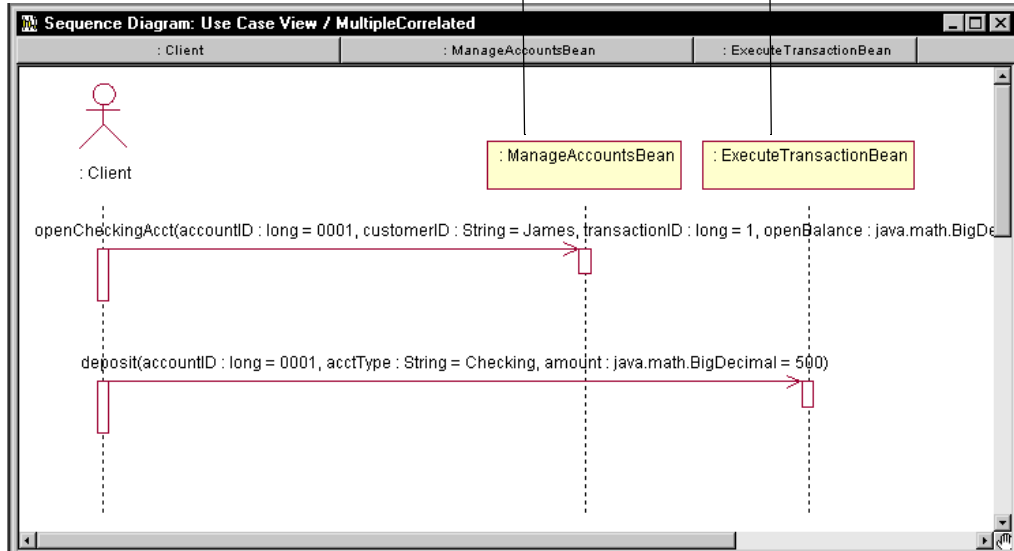
Scenario tests use Rose interaction diagrams—that is, sequence and collaboration diagrams—to test transactions. QualityArchitect interprets the messages in the interaction diagram and generates test scripts for one or more receiver objects in the diagram. (The *receiver object* is the object that receives the message.)

Each message in a diagram corresponds to an operation of a class, either by direct linkage through the diagram or by name. Each argument in a message maps to a corresponding parameter object attached to the operation object of the class.

As it generates the test script, QualityArchitect allows you to insert an optional verification point for each message in the diagram. (A *verification point* is a functional testing construct used by a test script to verify specific behavior. For more information, see *Verification Points* on page 23.)

Receiver object for openAcct message

Receiver object for deposit message



Scenario test scripts include support for datapools, logging, and verification points. (A *datapool* is a set of records that you can use to drive a test script. For details, see *Datapools* on page 21.) The scenario test templates provide automatic support for datapools and logging, while verification points can be added during script generation.

The basic steps for creating a scenario test are as follows:

- 1 Create a sequence or collaboration diagram in Rose.
- 2 Select the objects involved in the transaction and add them to the diagram.
- 3 Add messages for the operations that implement the transaction.
- 4 Add parameter names and data to the messages (optional).

In this step, you can modify message syntax to permit automated data correlation. For more information, see *Message Signatures and Data Correlation* on page 11.

- 5 Select a scenario test template.
- 6 Generate the scenario test and insert verification points.

For more information, see *Using the Java Query Builder to Add Verification Points* on page 59 and *Using the OLE DB Query Builder to Add Verification Points* on page 99.

- 7 Edit datapool data.

For more information, see *Datapools in Scenario Tests* on page 21. For information about creating datapools for EJB unit tests, see *Creating a Datapool for the Unit Test* on page 49. For information about creating datapools for COM unit tests, see *Creating a Datapool* on page 93.

Support for Performance Testing

In addition to using scenario test scripts for functional testing, you can use them for performance testing—that is, to measure how long a transaction takes to complete. QualityArchitect includes scenario testing templates that have been designed specifically for this purpose. Simply uncomment the line in the template for the TSSMeasure class, which is used to measure performance, as shown in the following EJB example.

```
public void testMain(String[] args) {
    boolean fRetval = false;
    TSSMeasure tss = new TSSMeasure(); // used for measuring performance
    TSSDataPool dp = new TSSDataPool();
    int idPCount = 0;
```

For more information, see the documentation for the TSSMeasure class in the *Rational Test Script Services for Java* manual and the *Rational Test Script Services for Visual Basic* manual.

Support for Verification Points

Typically, components tested with QualityArchitect are transactional, and as such they make updates to a database. Therefore, verification must include a way to verify these database transactions. Because scenario tests specifically test these transactions, QualityArchitect provides database verification points to verify any changes to the underlying database. When you generate a scenario test, QualityArchitect prompts you to insert a database verification point for each message in your interaction diagram.

For overview information about verification points, see *Verification Points* on page 23. For more detailed information, see the documentation for the VerificationPoint class in the *Rational Test Script Services for Java* manual and the *Rational Test Script Services for Visual Basic* manual.

For detailed information about database verification points, see the documentation for the DatabaseVP class in the *Rational Test Script Services for Java* manual and the *Rational Test Script Services for Visual Basic* manual.

Templates for Scenario Test Generation

QualityArchitect includes several templates that are used to generate scenario tests for each supported component model or application server. The templates shown in the first row in the following table are the ones you can select for test script generation. The other templates are called by the template that you select. The following templates are included with QualityArchitect:

Template	Description
websphere_scenario.template weblogic_scenario.template sunj2ee_scenario.template com_scenario_script.template	Top-level templates for testing the sequence of methods specified in an interaction diagram. Each top-level template contains substitution variables for the method calls listed in the interaction diagram.
websphere_scenario_constructor.template weblogic_scenario_constructor.template sunj2ee_scenario_constructor.template com_scenario_constructor.template	Templates for building the constructor. For EJBs, the constructor builds the remote interface. For COM, the constructor builds the COM object.
scenario_java_method1.template scenario_java_method2.template com_scenario_operation.template	Templates for generating the method calls. These templates include functions for calling individual methods within the sequence and for calling functions such as <code>tssCommandStart</code> and <code>tssCommandEnd</code> , which are used for performance testing.
com_scenario_project	Template for the Visual Basic project file that is generated to test a COM scenario.
com_scenario_basmain.template	Template for the main Visual Basic program that calls the test program in the <code>.cls</code> file. Visual Basic requires a <i>main</i> program to begin execution.

Message Signatures and Data Correlation

Message signatures include all of the message text that appears in parentheses in a Rose interaction diagram.

In the following message, for example, the message signature is shown in bold:

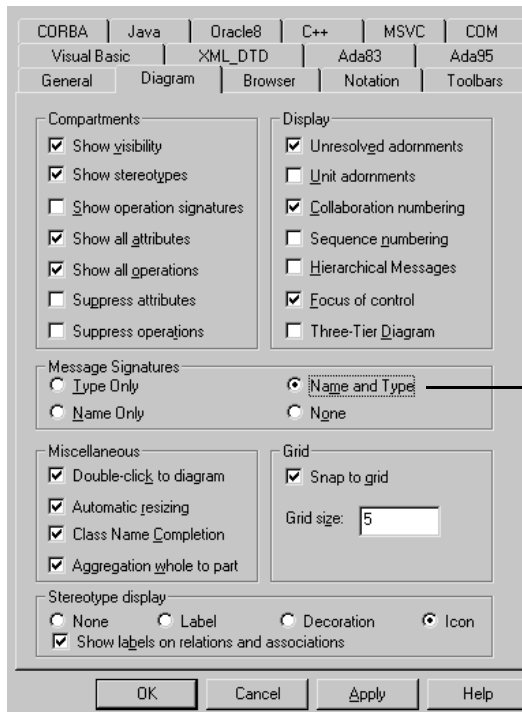
```
openCheckingAcct(accountID : long, customerID : String, transactionID : long, openBalance : java.math.BigDecimal)
```

Signature Options

Rose supports the following options for displaying message signatures:

- Type Only
- Name Only
- Name and Type
- None

You set these options in Rose by clicking **Tools > Options > Diagram** and clicking one of the check boxes inside the Message Signatures box.



Set message signatures here.

You can use message signatures for several purposes. For example, you can:

- Change parameter names to force the correlation results that you want.
- Add assignments to give initial values to parameters.
- Add assignments to give names to function results correlated with parameters.

Warning: If you change message signature options from one style to another in Rose—for example, from Name and Type to Type Only—every modification made to a manually edited message name is erased. Therefore, decide early on how you want message signatures displayed and do not change the options after you have modified the messages.

Message Syntax

When you generate a scenario test, QualityArchitect parses the diagram to determine the name, type, and initial data for each parameter (argument) in a message signature.

- **Parameter name.** For each parameter in the message signature, QualityArchitect uses the name from the message signature in the diagram. If no name is included in the message, QualityArchitect uses the name property of the associated parameter.
- **Parameter type.** For each parameter in the message signature, QualityArchitect uses the type from the associated parameter object, regardless of what is in the diagram.
- **Parameter's initial data.** For each parameter in the message signature, QualityArchitect uses the value to the right of the equals sign. If no initial data is included in the signature, QualityArchitect uses the Initial Data property of the associated parameter. If no Initial Data property is set and no initial data is specified in the diagram, then NULL or some other suitable value for the data type is used. Thus, you can select the **None** option in the Message Signatures box and still get a valid test script.

Note: When only the name or type of a parameter is present in the message, QualityArchitect compares this name or type to the actual type in the corresponding parameter object. If they are the same, QualityArchitect uses the name from the parameter object as the script variable name for the parameter. If they are different, QualityArchitect uses the name used in the message and takes the type from the parameter object.

Message syntax for the Name and Type option is as follows:

Result = operation([parameter₁ : type₁ = value₁, parameter_n : type_n = value_n, etc])

Item	Description
Result	The value (if any) returned by the operation. It is used by QualityArchitect for correlation with parameters of the same name later in the transaction. Result is optional.
Operation	The name of the operation as defined in a UML class.
Parameter Type Value	Parameter, type, and value define a parameter of the operation and its initial value. Parameter and type are both optional, because QualityArchitect examines the parameter object. The syntax that separates them with a colon is the UML syntax for separating parameter name and type. In QualityArchitect, you can add the assignment syntax (= value) to allow the initial value of a parameter to be specified in the sequence diagram.

Editing Message Signatures

You edit message signatures to tell QualityArchitect three things, all of which are optional:

- Parameter names that are different from their parameter specification
- Initial data
- The name and type of the return value

QualityArchitect generates a variable in the test script to hold the return value (result) if you do not supply one.

How Message Signatures Correlate to Datapool Fields

When you generate a scenario test, QualityArchitect also generates a corresponding datapool (see *Datapools in Scenario Tests* on page 21). QualityArchitect first examines the arguments in the message signatures to determine how many datapool fields to create. If no argument names are included in the signature, QualityArchitect supplies the names by searching the parameter (object) of the corresponding operation of the class.

For example, if QualityArchitect finds an argument named `accountID` in multiple messages, it creates a single datapool field for `accountID`. If you want to have multiple datapool fields for `accountID`, you need to assign `accountID` different names in the diagram, regardless of the Rose Message Signature Option that you have picked.

Message Signature Examples

This section provides several samples of message signatures.

Example 1 – the Name-Only Option

In this example, there is a `getBalance` operation in the `ExecuteTransaction` class. This operation takes one argument—`account ID`. If you want to generate a test script with two calls to this operation, with each call operating on different `account IDs`, you need to code your message signatures as follows:

```
getBalance(accountID1)
getBalance(accountID2)
```

In this example, two datapool fields will be created—one for `accountID1` and one for `accountID2`.

If you want these operations to go against the same account, driven from a datapool, you need to code your message signatures as follows:

```
getBalance(accountID)
getBalance(accountID)
```

The fact that both operations use the same argument name ensures that they are assigned to the same datapool field.

If you want to initialize these operations with data and assign them different names, code the message signatures as follows:

```
getBalance(accountID1 = 1)
getBalance(accountID2 = 5)
```

Example 2 – the Type-Only Option

If you use the Type-Only option, code your message signatures as follows to get the same datapool field for both calls:

```
getBalance( : Long)
getBalance( : Long)
```

In this case, the name of the datapool field will be taken from the parameter name of the parameter object.

If you want different datapool fields for each call, you must add the name to the signature, as in the following example, even if you have set Rose Options to Type-Only.

```
getBalance(accountID1 : long = 1)
getBalance(accountID2 : long = 5)
```

Example 3 – Name and Type

This example shows the message signature when you display name and type without including parameter values or assignment statements.

```
getBalance(acctountID : long, acctType : java.lang.String, amount :  
java.math.BigDecimal)
```

Example 4 – Name and Type with Parameter Values

This example shows the message signature after you add the initial data assigned to each parameter.

```
getBalance(acctountID : long = 1, acctType : java.lang.String =  
"checking", amount : java.math.BigDecimal = 50)
```

Example 5 – Name and Type with Assignment Statement

Finally, if the message returns a value and you want to establish a variable in the test script to hold this value, possibly for use as a parameter of data in a subsequent message, you need to convert the message name into an assignment statement. In this example, the variable named `result` is declared and used in the test script to hold the value returned by the `getBalance()` function.

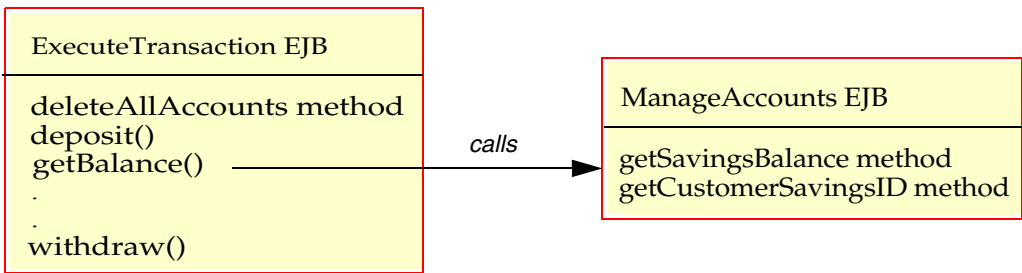
```
result = getBalance(acctID : long = 1, acctType : java.lang.String =  
"checking", amount : java.math.BigDecimal = 50)
```

Stubs

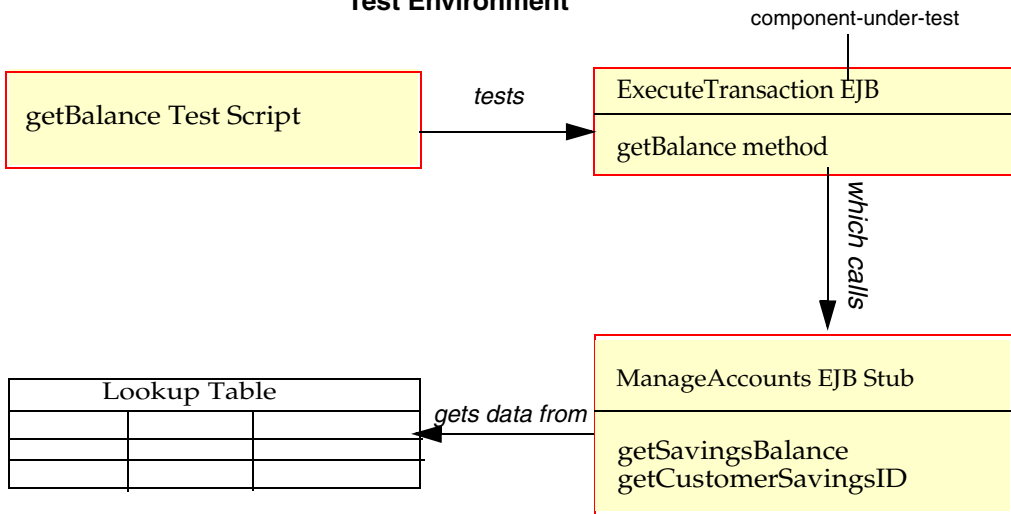
Stubs are components that can be used for testing purposes in place of actual components. A stub can be either a pure “dummy” component that just returns some predefined value, or it can simulate more complex behavior. With *QualityArchitect*, the simulated behavior for stubbed operations is specified in a lookup table. Through the use of stubs, you can control the results returned from components that interact with the component-under-test and create a simulated, controlled test environment.

Consider the sample EJB application that is described in the chapter on *Testing Enterprise JavaBeans*. In this chapter, you create a test script to test the `getBalance` method that is included with the `ExecuteTransaction` interface. Because the `getBalance` method calls several methods in the `ManageAccount` EJB, such as `getSavingsBalance` and `getSavingsCustomerID`, you would need to control access to the `ManageAccount` EJB to thoroughly test the `getBalance` method. As an alternative, you can create a stub for the `ManageAccount` EJB and control the results returned when `getBalance` calls the methods in the `ManageAccount` EJB.

Sample Application



Test Environment



In a stub, any method that is called by the component-under-test requires a lookup table consisting of rows and columns of expected results and exceptions for the method, as in the following example:

AccountID	expectedReturn	expectedException
012 34 5678	012 34 5678	
123 45 6789	123 45 6789l	
234 56 7890		java.rmi.RemoteException

Generate the table and columns with **Stubs > Create Look-up table**.

Use the Datapool Manager to populate the table.

The lookup table contains one or more columns for each parameter of the operation being stubbed. QualityArchitect uses the values in the columns to determine how the operation should behave. If the operation is supposed to return a value, the value is shown in the `expectedReturn` column. If the operation is supposed to throw an exception, the type of exception is shown in the `expectedException` column.

The stub looks up the row containing the inputs matching the parameter values passed in by the component-under-test—that is the caller of the stub—and either returns the value in the `expectedReturn` column or throws the exception in the `expectedException` column.

To generate stubs:

- 1 In the Rose browser, right-click the implementation class you are testing and click **QualityArchitect > Generate Stubs**.
- 2 Select a directory for storing the stubs.

For more information about stubs for EJBs, see *Generating Stubs for the Unit Test* on page 51. For more information about stubs for COM components, see *Generating Stubs for the Unit Test* on page 94.

To generate a lookup table:

- 1 Select a method called by the method that you are testing.
- 2 Click **Tools > QualityArchitect > Stubs > Create Look-up table**.

The lookup table is created with the name *ClassName_OperationName_L*—for example `ExecuteTransaction_getBalance_L`.

- 3 Use the datapool manager to populate the lookup table with data.

For information about populating datapools for EJBs, see *Creating a Datapool for the Unit Test* on page 49. For information about populating datapools for COM components, see *Creating a Datapool* on page 93.

For more information about lookup tables for testing EJBs, see *Creating Lookup Tables* on page 52. For more information about lookup tables for testing COM components, see *Creating Lookup Tables* on page 95.

Note: Stubs must be deployed on the same computer as the test script. Otherwise, playback will fail.

Templates for Stub Generation

EJB

The following tables list the templates that are used for EJB stub generation. The first table lists the templates used to generate stubs for the home interface, the remote interface, and the implementation class. The second table lists the templates used to build the method body in the EJB's implementation class. The template that is used depends on the particular method.

Template	Description
Session_Home.template	Template used to generate the stub for an EJB home interface.
Session_Remote.template	Template used to generate the stub for an EJB remote interface.
Session_Bean.template	Template used to generate the stub for an EJB implementation class.

Template	Description
MethodBodyWithoutLookUp.template	Template used when lookup code cannot be generated, either because: <ul style="list-style-type: none">▪ The method has no parameters▪ The method has no return value or exceptions▪ The method contains at least one complex parameter and lookup code cannot be generated automatically.
MethodBodyWithoutExceptions.template	Template used when the method throws no exceptions.
MethodBodyWithoutReturnValue.template	Template used when the method has no return value (for example, returns void).
MethodBody.template	Template used for all other methods.

COM/VB

The following tables list the templates that are used for COM/VB stub generation. The first table lists the template used to generate the code that defines the Visual Basic class. The second table lists the templates that are used to create the method bodies for the COM/VB stub. Different templates are used depending on the type of method being stubbed.

Template	Description
VBCOMClass.template	Template used in all cases to generate the code that defines the Visual Basic class.

Template	Description
FunctionBody.template	Template used for functions. (Functions are methods that have return values.)
FunctionBodyWithoutLookUp.template	Template used for functions without parameters. (No look up code is generated when a method has no parameters.)
PropertyGetBody.template	Template used for "Property Get" methods.
PropertyGetBodyWithoutLookUp.template	Template used for "Property Get" methods without parameters. (No lookup code is generated when a method has no parameters.)
PropertyLetBody.template	Template used for "Property Let" methods.
SubBody.template	Template used for subroutines. (Subroutines are methods that have no return value.)
SubBodyWithoutLookUp.template	Template used for subroutines without parameters. (No lookup code is generated when a method has no parameters.)

Test Script Services

Rational Test Script Services provide datapool, logging, verification, synchronization, measurement, and monitoring capabilities to various Rational applications, including QualityArchitect.

Datapools

A *datapool* is a set of records that you can use to drive a test script. Typically, each record in the datapool represents a test case and includes either the test inputs, the expected results, or both. With a datapool, a single script can iterate through multiple test cases. If you want to add a new test case, such as the result of invalid input, you only need to add another record to the datapool.

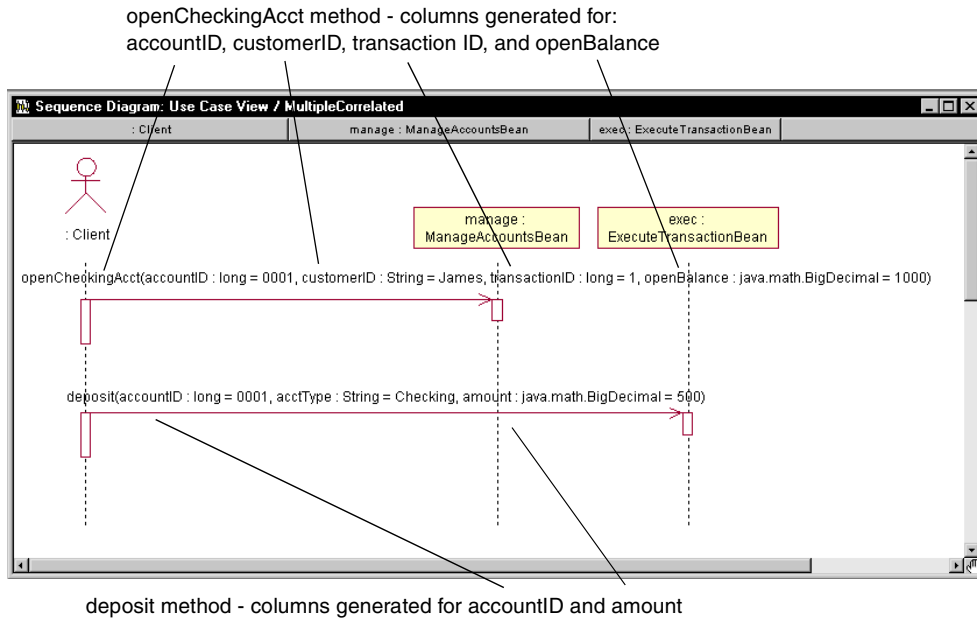
For example, suppose you want to test an `add()` operation that takes two integers as parameters. If you want to test what happens when you pass it two negative integers, you can simply add another record to the datapool. Include negative integer values in each of the two parameter columns, and populate the expected result or expected exception columns accordingly.

Datapools greatly reduce the number of test scripts that are required and minimize script maintenance.

Datapools in Scenario Tests

When you generate a scenario test, QualityArchitect creates a datapool automatically. It examines each message in an interaction diagram and populates one datapool row with data based on the parameter values in the messages. If no parameter values are found in the messages, QualityArchitect attempts to populate the row with initial values found in the parameter specification.

QualityArchitect creates a datapool column for each unique parameter in each message in the diagram. For example, as shown in the following figure, if the first message included in the script has four unique parameters, and the second message has two unique parameters, the datapool will have at least six columns. In addition, each verification point contributes several columns of data to the datapool. The initial data for these verification point columns is extracted from the values entered into the Query Builder wizard.



By default, QualityArchitect assigns the datapool a name of *ModelName_DiagramName__D*.

You can access and edit the datapool from the Rational QualityArchitect console. To display the console, click **Tools > QualityArchitect > Console**.

Datapools in Unit Tests

Datapools are not created automatically when you generate a unit test script. However, you can create a datapool very easily. To do so:

- Right-click an operation in the Rose browser and click **QualityArchitect > Create Datapool**.

QualityArchitect creates the new datapool and presents you with a dialog box in which you can add rows of data. QualityArchitect uses the operation's parameters as datapool columns and names the datapool *ClassName_OperationName__D*.

Note: Because the datapool name and its column names are embedded in the script, you will need to modify your script if you change the datapool name.

For an EJB example, see *Creating a Datapool for the Unit Test* on page 49. For a COM example, see *Creating a Datapool* on page 93.

For detailed information on the use of datapools with other Rational Test products, see the Help for Rational TestManager and Rational Robot.

Data Types

Data types are used to define datapool columns. You assign data types to datapool columns when you define the columns in the Datapool Specification dialog box.

For Java, QualityArchitect supports all native data types plus `String` and `BigDecimal`.

For Visual Basic, values are retrieved as variants containing strings.

For a complete list of available data types, see the TestManager Help.

Note: Complex data types are not currently supported in datapools. However, you can construct complex data types at runtime based on the primitive data types that the datapool supplies.

Verification Points

A verification point (VP) is a functional testing construct used by a test script to verify specific behavior in the application or component-under-test. In QualityArchitect, a verification point compares an *expected* data object with an *actual* data object. The result of this comparison is logged, allowing for analysis of overall functional correctness and test case coverage.

How Data Is Verified

A verification point operates on two different types of data:

- Data that is known to be correct.

For example, this data might be captured when the component is known to be functioning correctly, or from a source that is known to contain the correct data.

Data that is known to be correct is called the *expected* data.

- Data whose validity is unknown and must be verified.

This data is captured at test runtime and is called the *actual* data.

A verification point compares expected data and actual data. If the data matches (or optionally, satisfies some other condition, such as falling within an accepted range), the verification point passes. Otherwise, the verification point fails. Verification point results are automatically logged and viewed in the Rational Test Log window.

Static, Dynamic, and Manual Verification Points

In QualityArchitect there are three types of verification points: static, dynamic, and manual.

Static Verification Points

With a static verification point, the expected data object is captured the first time the verification point is executed and is stored in the datastore as the verification point's baseline. The expected data object does not change during subsequent executions of the test script. When the verification point is executed again:

- An actual data object is captured from the system-under-test.
- The baseline data object is retrieved from the datastore.
- The two objects are compared, and the result is logged.

Static verification points are regression-style tests in that the successful behavior of the system is implicitly defined by the system's behavior the first time the verification point executes. In other words, static verification points assume that the system-under-test is functionally correct the first time you run the verification point.

QualityArchitect contains a user-acknowledgement flag that lets you interactively accept or reject data as a baseline for a static verification point. With the flag enabled, QualityArchitect will prompt you with the baseline data that you can accept or reject. For details, see *Verification Point classes* in the online *Rational Test Script Services for Java* and *Rational Test Script Services for Visual Basic* manuals.

To access the online documentation:

- Click **Start > Programs <Rational ...> Rational Test > API > TSS for Java** or
- Click **Start > Programs <Rational ...> Rational Test > API > TSS for Visual Basic.**

where *<Rational ...>* is the name of the Rational product you have installed—for example, Rational Suite DevelopmentStudio.

Dynamic Verification Points

With a dynamic verification point, the expected data object is passed to the verification point at execution time. This expected data object is not managed by the verification point infrastructure and may be hard coded into the script, driven by values in a datapool, or built using any other method chosen by the test script author. When a dynamic verification point is executed, an expected data object is passed to the verification point by the test script. The verification point captures an actual data object from the system-under-test, compares the expected and actual data objects, and logs the result. Dynamic verification points differ from static verification points in that the successful behavior of the system under test is explicitly defined by the test script author, not implicitly defined by a previous behavior of the system-under-test.

Manual Verification Points

With manual verification points, you as the scripter are responsible for providing both the expected and the actual data objects. The verification point framework simply compares the data objects that you provide and logs the results.

Database Verification Point

In this release, Rational has implemented a single verification point: the database verification point. This verification point is used to validate the changes made to a data source by the component-under-test. If you are testing a component that makes modifications to a data source, you can invoke a method on that component and then use a database verification point to verify that the expected changes to the data source actually occurred.

With a static database verification point, the first time that you run a test, the data is captured and stored in the datastore, thus establishing a baseline. Subsequent runs will show a pass in the Rational Test Log window if the returned values are the same as the values in the baseline. With a dynamic database verification point, the test script itself constructs the expected value.

The Query Builder

QualityArchitect provides a Query Builder wizard that lets you build and execute queries for use with a database verification point. Wizards are available for both EJB and COM. The wizards let you specify a data source and build a query.

The Query Builder appears automatically in the following situations:

- When you insert a database verification point while generating a scenario test
- When you insert a database verification point from the EJB Session Recorder (EJB only)
- When you run a test containing an incompletely-specified database verification point

Extensibility

QualityArchitect provides you with the ability to implement your own verification points by extending the verification point framework classes. For details, see *Implementing a New Verification Point* in the *Rational Test Script Services for Java* and *Rational Test Script Services for Visual Basic* manuals.

This chapter contains the information that you need to use Rational Quality Architect to test Enterprise JavaBeans (EJBs). Topics include:

- Overview
- Requirements for EJB testing
- The Rational Bank Account Sample Application
- Reverse engineering a deployed EJB into Rational Rose
- Generating test assets
- Importing test assets into IBM Visual Age for Java
- Executing test scripts
- Using scenario tests to test transactions
- Using the Query Builder to add verification points
- Using the EJB Session Recorder

Overview

This section provides a brief overview of EJB development and test.

For each EJB that you develop, you must define two interfaces and either one or two classes—the home interface, the remote interface, the implementation class, and the primary key class.

- The *remote interface* defines all of the bean's business methods.
- The *home interface* defines lifestyle methods for the bean—that is, methods for creating, removing, and finding beans.
- The *implementation* or *bean class* implements the business methods defined by the remote interface.
- The *primary key class* provides a pointer into a database. Only *entity beans* require a primary key.

Because client applications interact directly with the home and remote interfaces and not with the implementation class, and because test scripts emulate client applications, EJB testing needs to test only the methods defined in the remote and home interfaces.

EJB TestScripts

All test scripts that are generated or constructed manually extend `com.rational.test.tss.TestScript`, an abstract class that contains the code necessary to connect to a project, initialize log information, and look up the name of the script.

All EJB test scripts that extend the `TestScript` class must implement a `main` method and a `testMain` method.

- The `main` method is the entry point for the class.
- The `testMain` method is the entry point for the testing code. `testMain` includes all of your test logic. All calls to other test scripts must reside within `testMain`.

In addition, all EJB test scripts must call `tms.startTestServices` and `tms.endTestServices`.

- `tms.startTestServices` initializes logging and other test services and should be the first method called in `testMain` (within a try block).
- `tms.endTestServices` turns off logging, writes the log file, and performs various cleanup functions and should be the last method called in `testMain` (in your finally block).

Package Requirements

EJB test scripts must also import the following packages:

- `com.rational.test.ct.*`
- `com.rational.test.tss.*`
- `com.rational.test.vp.*`
- `com.rational.test.vp.ui.*`
- `java.util.*`
- `javax.naming.*`

In addition, any test script to be deployed in a WebSphere environment must import the following package:

- `javax.rmi.PortableRemoteObject`

For more information about the TestScript class, see the online *Rational Test Script Services for Java* manual. (Rational Test\Api\TssJava\Rational TSS for Java.htm).

Requirements for EJB Testing

This section describes the following configuration requirements for EJB testing:

- Supported environments
- Test script playback requirements, including requirements for running test scripts from Rational TestManager
- VAJ/WebSphere requirements
- Visual Cafe/WebLogic requirements

See *EJB Session Recorder Requirements* on page 67 for additional requirements for the EJB Session Recorder.

Supported Environments

The following tables list the JDKs and protocols supported for each environment:

SUN JDK/Sun J2EE Reference Application Server v.1.2.1

Protocol	JDK 1.18	JDK 1.2x	JDK 1.3x
JNDI	N/A	N/A	Yes
RMI/IIOP	N/A	N/A	Yes

Visual Cafe 3.1/WebLogic 4.5.x

Protocol	JDK 1.18	JDK 1.2x	JDK 1.3x
JNDI	Yes	Yes	N/A
RMI/IIOP	N/A	N/A	N/A

Visual Cafe 4.0/WebLogic 5.1

Protocol	JDK 1.18	JDK 1.2x	JDK 1.3x
JNDI	Yes	Yes	N/A
RMI/IIOP	N/A	N/A	N/A

VAJ 3.02/WebSphere Runtime

Protocol	JDK 1.18	JDK 1.2x	JDK 1.3x
JNDI	N/A	N/A	N/A
RMI/IIOP	Yes	Yes	N/A

VAJ 3.5/WebSphere Runtime

Protocol	JDK 1.18	JDK 1.2x	JDK 1.3x
JNDI	N/A	Yes	N/A
RMI/IIOP	N/A	Yes	N/A

The installation procedure includes several packages that are required for test script playback. These packages are installed in the form of JAR files and must be added to your IDE's classpath to enable script playback from your IDE.

The required packages include:

Package	File Name	Default Installation Path
Rational Test Script Services	rational_ct.jar rttssjava.jar	C:\Program Files\Rational\Rational Test\QualityArchitect and C:\Program Files\Rational\Rational Test
Rational Test Script Execution Adapter	rttseajava.jar	C:\Program Files\Rational\Rational Test\tsea
Swing v.1.1.1	swingall.jar	C:\Program Files\Rational\Rational Test\QualityArchitect
JavaHelp v.1.1	jh.jar	C:\Program Files\Rational\Rational Test\QualityArchitect

Note: JavaHelp is required to display online Help for the EJB Query Builder and the EJB Session Recorder. If this JAR file is not included in the IDE's classpath, a `NoClassDefFoundError` will be generated when the Test Script is executed or when the Query Builder is launched.

Rational TestManager Playback Requirements

You can also play back your test scripts directly from Rational TestManager. In order to do so, you must include:

- The full path to a Java compiler on your system path
- The full path to a JVM on your system path if you are running a pre-1.2.x version of the JVM

Note: With a standard 1.2x or 1.3 JDK installation, Rational will attempt to locate the JVM automatically relative to the location of the compiler (javac.exe).

- The class files for your client application on your classpath
- Any required application server classes on your classpath

In addition, if TestManager needs to connect to a WebSphere application server, the following JAR files must be added to your classpath:

- ejs.jar
- ibmjndi.jar
- ns.jar
- ujc.jar

Visual Age/WebSphere Requirements

In order to run the sample application, you need:

- Visual Age for Java 3.02 (VAJ) or newer
- The VAJ WebSphere Test Environment and VAJ EJB Development Environment
For details about how to load these features, see the VAJ online Help.
- DB2 version 6.1 with fix pack 2

You can download FixPak 2 from the following location:
<http://www.ibm.com/DB2>.

Visual Cafe/WebLogic Requirements

In order to run and deploy the sample application in a Visual Cafe/WebLogic environment, you need:

- Visual Café 3.1 or newer
- WebLogic 4.5.x or newer

In addition, the Rational Quality Architect installation installs the following JAR files—`ratlbankacct45.jar` and `ratlbankacct51.jar`—and a CloudScape database, both of which are needed to run the sample application.

Visual Cafe/WebSphere Requirements

In order to allow Visual Cafe to run test scripts that are deployed on a WebSphere application server, you must add the IBM JRE to the list of virtual machines in Visual Cafe. To do this, click **Tools > Environment Options > Virtual Machines** in Visual Cafe.

EJB Session Recorder Requirements

For information about requirements for the EJB Session Recorder, see *EJB Session Recorder Requirements* on page 67.

The Rational Bank Account Sample Application

The Rational Bank Account sample application is a Java-based client that you can use to try out the features in Quality Architect. With the Rational Bank Account sample application, you can set up bank accounts and make deposits and withdrawals. The application consists of the following items:

- A Java-based client application
- A set of six EJBs—four entity beans and two session beans

Each entity bean corresponds to a different table in a database and uses *container managed persistence*. The session beans are stateless and are used to interact with the client; the entity beans are used to manage transactions with the database.

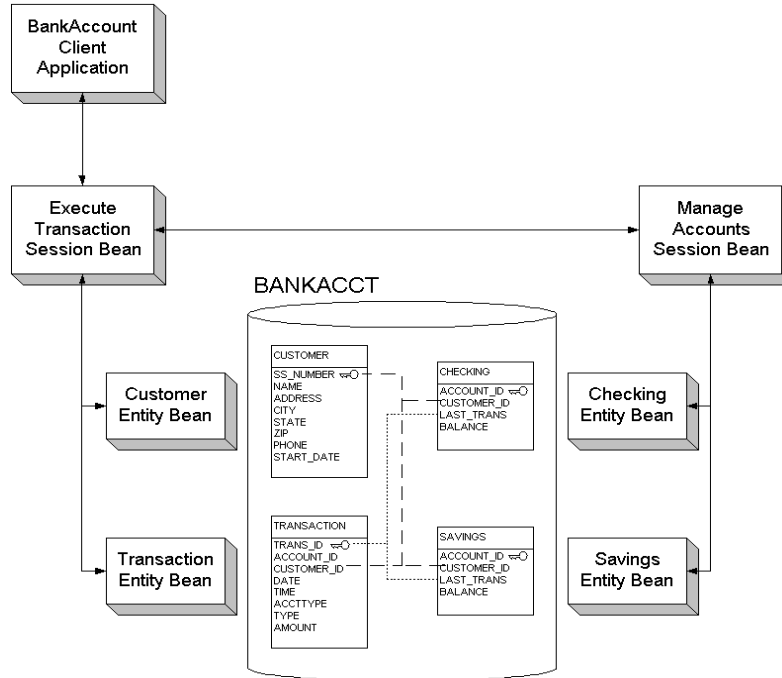
- A Rose model—`ratlbankacct.mdl`—that you can use to generate test scripts and stubs against the sample EJBs that are provided.

This model is populated with objects derived from reverse engineering the sample application's deployed JAR file. It uses the J2EE modeling support built into Rose J to model the beans in the sample application and implements a standard profile for stereotypes and associations for modeling EJBs in UML. For more information about J2EE modeling support, see *How Rational Rose Supports the Java Platform Enterprise Edition (J2EE)* in the Rose J Online Help.

Note: The procedures that are used to demonstrate EJB testing with Quality Architect are all based on the sample application. For example, one procedure shows you how to generate a unit test for the `getBalance` method in the `ExecuteTransaction` remote interface, while another procedure shows you how to create a stub for the methods that are called by `getBalance` method.

The following figure shows the structure of the Rational Bank Account sample application.

Enterprise JavaBeans Bank Account Sample



Setting Up the Sample Application for VAJ/WebSphere

This section shows you how to set up the sample application to run in a VAJ/WebSphere environment.

Importing the Sample into VAJ

Before you can run the sample application, you must import it into VAJ, as follows:

- 1 Start IBM Visual Enterprise for Java (VAJ).
- 2 Within the VAJ Workbench window, click the **Projects** tab.
- 3 Right-click anywhere in the window and click **Import**.

- 4 When the SmartGuide dialog box appears, select **Repository** as the import source and click **Next**.
- 5 Select **Local Repository**, click the **Browse** button, and navigate to the *<install directory>* Program Files\Rational\Rational Test\QualityArchitect\Samples\ejb\bankacct\vaj directory.
- 6 For VAJ 3.0x, select RatlBankAcct30.dat and click **Open**. For VAJ 3.5, select RatlBankAcct35.dat and click **Open**.
- 7 Make sure **Projects** is selected in the 'What do you want to import?' section.
- 8 Click **Details**.
- 9 Select **Rational Bank Account Sample** in the Projects list.
 - For VAJ 3.0x, make sure **1.2.1** is selected in the Versions window.
 - For VAJ 3.5, make sure **1.2.2** is selected in the Versions window.
- 10 Click **OK**.
- 11 Make sure **Add most recent project edition to workspace** is checked.
- 12 Click **Finish** to import the sample into the repository.

If you cannot see the Rational Bank Account Sample project in the Projects window:

- 1 Right click in the window and select **Add > Project**.
- 2 Select **Add projects from the repository**.
- 3 Select **Rational Bank Account Sample** from the Available project names list.
 - For VAJ 3.0x, make sure **1.2.1** is selected in the Versions window.
 - For VAJ 3.5, make sure **1.2.2** is selected in the Versions window.
- 4 Click **Finish**.

You should now see the Rational Bank Account Sample project in the Projects window.

Creating the Database

To run the sample, you must first create a database in DB2 and then configure VAJ to use the database. Because of limitations in VAJ, the database must reside on the same server as the sample application.

To create the database:

- 1 From the Start menu, click **Programs > Db2 for Windows NT > Command Line Processor** to launch the DB2 Command Line Processor.
- 2 At the db2=> prompt, type **create database** *<database name>*, where *<database name>* is the actual name of the database that will be used for the sample—for example, **create database bankacct**.

You should see a return message in the window that states the database was created successfully.

- 3 Close the Command Line Processor window by typing **quit** and then **exit** at the db2=> prompt.

Configuring VAJ to Use the New Sample Database

To configure VAJ to use the new sample database:

- 1 Launch VAJ.
- 2 Specify the DB2 JDBC driver to use.
 - a From the Workbench, click **Window > Options**.
 - b In the Options dialog box, select **Resources** and enter the following information in the Workspace class path text box:

`<DB2 Path>\java\db2java.zip`

where *<DB2 Path>* is the full DB2 installation path and db2java.zip is the DB2 JDBC driver. The default DB2 installation directory is SQLLIB.

If you do not know where DB2 is installed, search for the file name db2java.zip to determine the full file path. The following is an example of the JDBC driver path:

`D:\SQLLIB\java\db2java.zip`

- c Click **Apply** to save the selected DB2 JDBC driver information, and then click **OK**.

Adding EJB Support

To add EJB support:

- 1 Launch VAJ.
- 2 Click **File > QuickStart** from the main VAJ menu.
- 3 Click **Features** in the left-hand box and **Add Features** in the right-hand box.

- 4 Click **OK**.

A dialog box appears with a list of features.

- 5 Click **IBM EJB Development Environment** and **IBM WebSphere Test Environment** and click **OK**.

To make sure that these features were added successfully, make sure that the EJB tab is visible in the Workbench window.

Deploying the Sample Application in the VAJ Test Environment

This section includes instructions that apply to VAJ 3.0x and VAJ 3.5.

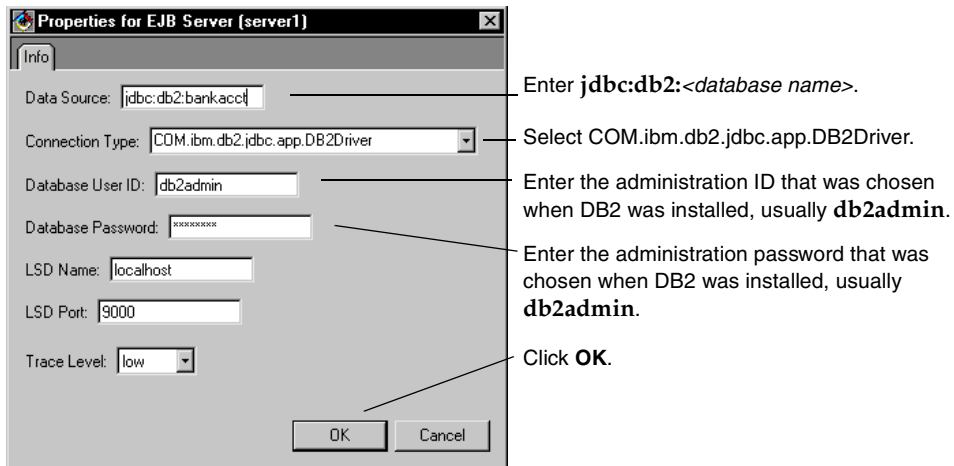
Creating Database Tables (VAJ 3.0x and VAJ 3.5)

- 1 In VAJ, click the **EJB** tab to view the EJB window.

You should now see the RationalBankAcctSampleEJBs group in the window.

- 2 Right-click **RationalBankAcctSampleEJBs** and click **Add To > Server Configuration**.

- 3 Right-click **EJB Server (server 1)** and click **Properties**.



- 4 Right-click **EJB Server (server 1)** again and choose **Create Database Table**.

This creates the database tables that are required for the sample.

At this point, the deployment process varies, depending on your version of VAJ. See either *Starting the Location and Name Servers (VAJ 3.0x Only)* on page 37 or *Starting the Servlet Engine and the Name Server (VAJ 3.5 Only)* on page 37.

Starting the Location and Name Servers (VAJ 3.0x Only)

- 1 Right-click **Location Service Daemon** and click **Start Server**.

The Location Service Daemon is used to facilitate the WebSphere Test Environment.

- 2 Right-click **Persistent Name Server** and click **Start Server**.

The Persistent Name Server is the JNDI server that is required for the naming service.

The VAJ Console window should appear. You will know that the server has been successfully started when the following line appears in the Output box:

```
EJServer E Server open for business
```

Starting the Servlet Engine and the Name Server (VAJ 3.5 Only)

- 1 From the Workbench window, click **Workspace > Tools > WebSphere Test Environment**.

This opens the WebSphere Test Environment Control Center.

- 2 Click **Servlet Engine** and then click **Start Servlet Engine**.

The VAJ Console window should appear. You will know that the server has been successfully started when the following line appears in the Output box:

```
Servlet Engine is started
```

- 3 Click **Persistent Name Server** and click **Start Name Server**.

The Persistent Name Server is the JNDI server that is required for the naming service.

You will know that the server has been started when the following message appears at the bottom of the WebSphere Test Environment Control Center:

```
Persistent Name Server is Started
```

Starting the EJB Server (VAJ 3.0x and VAJ 3.5)

From this point on, the instructions are the same for VAJ 3.0x and VAJ 3.5.

- 1 Back in the EJB Server Configuration window, right-click **EJB Server (server 1)** and click **Start Server**.
- 2 Click on the Console window to bring it to the front.

Note: It might take a moment for the Console window to be brought to the front.

- 3 In the **All Programs** box, click the last EJB Server entry.

You will know that the server has been successfully started when the following line appears in the Output box:

```
EJServer E Server open for business
```

Starting the Rational Bank Account Sample Application (VAJ 3.0x and VAJ 3.5)

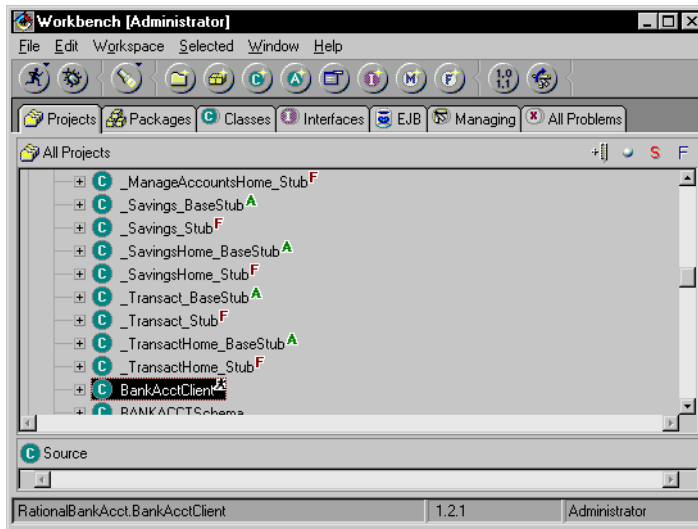
- 1 Click the **Projects** tab in the VAJ Workbench window.
- 2 Find the Rational Bank Account Sample in the window and expand it by clicking on the **+** box.

If there is a - in the box, then it is already expanded.

- 3 Expand the RationalBankAcct package by clicking the **+** box.

If there is a - in the box, then it is already expanded.

- 4 Scroll until you see the BankAcctClient class. (On the right side of the class there should be a picture of a running man.)



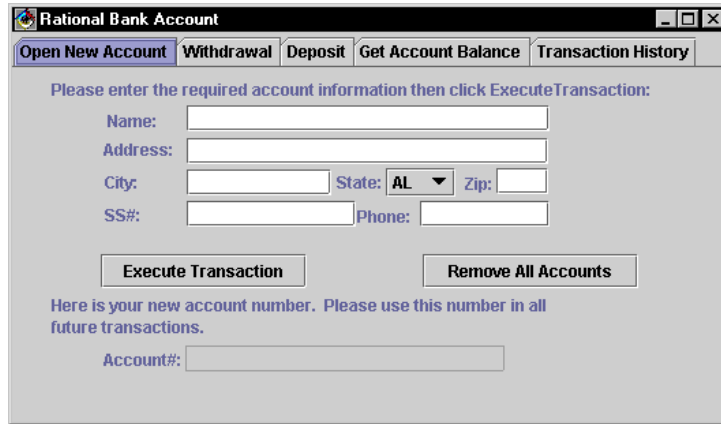
- 5 Right-click on this class and click **Run > Check Classpath**.
- 6 Click **Compute Now**.
- 7 If you plan to use the WebSphere Test Environment, click the **Edit** button to the right of the Project path and make sure the IBM WebSphere Test Environment package is checked. Click **OK**.
- 8 Click the Program tab.

In the Command line arguments box, enter the following text:

```
appserver=websphere
```

9 Click **OK**.

10 Right-click on this class and click **Run > Run main**.



At this point, the Rational Bank Account window appears in the upper left corner of your screen. You will use this client application to add database entries that you can use for testing purposes. For details, see *Adding Account Information to the Rational Bank Account Sample* on page 44.

Importing Swing and the Rational Support Classes

Before importing generated test scripts into VAJ, you must import the Swing foundation class library (`swingall.jar`) and the Rational support classes (`rational_ct.jar`, `rttseajava.jar`, and `rttssjava.jar`). Swing is required for various UI components, such as the Query Builder and the EJB Session Recorder. The Rational support classes provide support verification points, datapools, stub generation, and other services. These JAR files must be added to VAJ's classpath to enable script playback from VAJ. You can find these files in the following locations:

- `<Install_Dir>\Rational Test\QualityArchitect\swingall.jar`
- `<Install_Dir>\Rational Test\QualityArchitect\rational_ct.jar`
- `<Install_Dir>\Rational Test\tsea\rttseajava.jar`
- `<Install_Dir>\Rational Test\rttssjava.jar`

To import Swing and the Rational support classes:

- 1 Right-click on the Rational Bank Account Sample project and click **Import**.
- 2 Make sure that the JAR file option is selected and click **Next**.

- 3 Under **What types of files do you want to import**, make sure that the **.class** option and the **resource** option are checked.
- 4 Under **Options**, make sure that **Create new/scratch editions of versioned projects/packages** is checked.
- 5 Click the **Browse** button and navigate to the installation directory for QualityArchitect (Rational\Rational Test\QualityArchitect).
- 6 Select swingall.jar and click **Open** in the File Open dialog box.
- 7 Click **Finish**.

All of the class files from the JAR file will then be imported into the VAJ repository.

After **swingall.jar** is imported, an importing problems dialog box is displayed. The dialog box essentially states that the following Macintosh Swing class cannot be changed:

```
com.sun.java.swing.plaf.mac.MacBorders.class
```

You can ignore these problems because they will not affect the EJB Session Recorder.

- 8 Click **OK** to close the problems dialog box.
- 9 Repeat these steps for the Rational support classes.

Deploying the Sample Application in WebLogic

This section describes how to deploy the sample application to WebLogic 4.5.x and WebLogic 5.1.

To deploy the sample application in WebLogic:

- 1 Create a subdirectory within the *<weblogic directory>\myserver* directory called `RatlBankAcct`.
- 2 Copy the JAR files.
 - For WebLogic 4.5.x, copy the `RatlBankAcct45.jar` file from the `Rational Test\QualityArchitect\Samples\ejb\bankacct\vc` directory to the *<weblogic directory>\myserver\RatlBankAcct* directory that was created in the Step 1.
 - For WebLogic 5.1, copy the `RatlBankAcct51.jar` file from the `Rational Test\QualityArchitect\Samples\ejb\bankacct\vc` directory to the *<weblogic directory>\myserver\RatlBankAcct* directory that was created in the Step 1.
- 3 For WebLogic 4.5.x, copy the `bankacct` directory from `Rational Test\QualityArchitect\Samples\ejb\bankacct\vc\data\weblogic45` to *<weblogic directory>\eval\cloudscape\data*.

For WebLogic 5.1, copy the bankacct directory from Rational Test\QualityArchitect\Samples\ejb\bankacct\vc\data\weblogic51 to <weblogic directory>\eval\cloudscape\data.

These directories contain the Cloudscape databases for the sample. (For details, see the Cloudscape readme file.)

- 4 Open the weblogic.properties file, found in <weblogic directory>, for editing. Uncomment (by removing the # symbol from the beginning of the line) the following lines found in the WEBLOGIC DEMO CONNECTION POOL PROPERTIES section:

```
weblogic.jdbc.connectionPool.demoPool=\
url=jdbc:cloudscape:demo,\
driver=COM.cloudscape.core.JDBCdriver,\
initialCapacity=1,\
maxCapacity=2,\
capacityIncrement=1,\
props=user=none;password=none;server=none
weblogic.allow.reserve.weblogic.jdbc.connectionPool.demoPool=everyone
```

- 5 Change the following lines:

From	url=jdbc:cloudscape:demo,\
To	url=jdbc:cloudscape:bankacct,\
From	weblogic.jdbc.connectionPool.demoPool=\
To	weblogic.jdbc.connectionPool.rqaPool=\
From	weblogic.allow.reserve.weblogic.jdbc.connectionPool.demoPool=everyone
To	weblogic.allow.reserve.weblogic.jdbc.connectionPool.rqaPool=everyone

- 6 Add the following lines:

- For WebLogic 4.5.x, add the following lines right after the WEBLOGIC EJB DEMO PROPERTIES section:

```
weblogic.ejb.deploy=\
<weblogic directory>/myserver/RatlBankAcct/RatlBankAcct45.jar
```

- For WebLogic 5.1, add the following lines right after WEBLOGIC EJB DEMO PROPERTIES section:

```
weblogic.ejb.deploy=\
    <weblogic directory>/myserver/RatlBankAcct/RatlBankAcct51.jar
```

7 Save the weblogic.properties file.

8 Launch the WebLogic server:

For WebLogic 4.5.x, click **Start > Programs > WebLogic 4.5.x > WebLogic Server**.

If the Rational Bank Account sample was successfully deployed, you should see the following line in the WebLogic Server window:

```
<EJB> 1 EJBs were deployed using .jar files
```

For WebLogic 5.1, click **Start > Programs > WebLogic 5.1 > WebLogic Server**.

If the Rational Bank Account sample was successfully deployed, you should see the following lines in the WebLogic Server window:

```
<EJB> 1 EJB jar files loaded, containing 6 EJBs
```

```
<EJB> 6 deployed, 0 failed to deploy.
```

Note: If the EJBs do not deploy, check the following file:

```
<installdir>\eval\cloudscape\data\cloudscape
```

Deploying the Sample Application to the Sun J2EE Reference Server

To deploy the sample application to the Sun J2EE Reference Server, you must first have the latest J2EE SDK v.1.2.1 installed. Throughout this section, %J2EE_HOME% is used to represent the currently installed location for the J2EE SDK.

To deploy the sample application to the Sun J2EE Reference Server:

1 Configure the data source that the J2EE SDK will use for the Bank Account sample.

a Open the %J2EE_HOME%\config\default.properties file.

b Find the line for the property jdbc.datasources. At the end of the line, append the following text:

```
| jdbc/BankAcctDB| jdbc:cloudscape:rmi:CloudscapeDB;create=true
```

2 Start the Cloudscape database server.

In a separate command prompt window, change the directory to %J2EE_HOME%\bin and type the following text:

```
cloudscape -start
```

3 Start the J2EE Reference Server.

In a separate command prompt window, change the directory to %J2EE_HOME%\bin and type the following text:

```
j2ee -verbose
```

4 Launch the Sun Deployer tool that is used to deploy the sample application to the Sun J2EE Reference Server.

a In a separate command prompt window, change the directory to %J2EE_HOME%\bin and type the following text:

```
deploytool
```

b In the Sun Deployer tool application window, click **File > Open Application**.

c In the Open File dialog box, browse to the <RQA_INSTALL>\Samples\EJB\BankAccount\J2EE directory and select the RatlBankAcctEE.EAR file.

d Click **Tools > Deploy Application**.

e Continue through the dialog boxes and accept all of the default settings by clicking **Next**.

f When you reach the last dialog box, click **Finish**.

The sample EJB application will now be deployed to the Sun J2EE Reference Server.

Configuring Visual Cafe to Run the Sample Application

To run the Rational Bank Account sample EJB application in Visual Café:

1 Launch Visual Café 3.1 if it is not currently running.

2 Create a new Empty Project:

a Click **File > New Project**.

b Select **Empty Project**.

c Click **OK**.

3 Click **Project > Options** and then click the **Directories** tab.

4 Click **New**, the button furthest to the left.

5 Click **file**.

- 6 For WebLogic 4.5.x, navigate to Rational Test\QualityArchitect\Samples\ejb\bankacct\vc\RatlBankAcct45.jar and click **Open**.

For WebLogic 5.1, navigate to Rational Test\QualityArchitect\Samples\ejb\bankacct\vc\RatlBankAcct51.jar and click **Open**

- 7 Click the **Project** tab.
- 8 In the Main Class box, type **RationalBankAcct.BankAcctClient**.
- 9 In the Program Arguments box, enter the following text:
`appserver=weblogic`
- 10 Click **OK**.

After this point, you can add account information and configure datapools and stub lookup tables.

Adding Account Information to the Rational Bank Account Sample

After the Rational Bank Account window opens, you need to create a few savings accounts and make deposits to each account. Be sure to keep track of the generated account numbers and the balance in each account. You will need this information later when you create the datapool.

To create a new account:

- 1 Click **Open New Account**.
- 2 Enter the required account information and click **Execute Transaction**.
- 3 Write down the number of the new account.

To make a deposit:

- 1 Click **Deposit**.
- 2 Enter the account number that was returned when the account was created.
- 3 Enter the amount you want to deposit.
- 4 Select **Savings**.
- 5 Click **Execute Transaction**.

Be sure to keep track of the account balance.

Reverse Engineering a Deployed EJB into Rose

The sample model that is included with the Rational Bank Account sample application is populated with objects derived from reverse engineering the sample application's deployed JAR file. You can use this model to generate test assets to test the sample application. For details, see *Generating Test Assets* on page 45.

If you already have your own deployed EJBs that you want to test, you can reverse engineer them into Rose. When you reverse engineer the deployed JAR file, Rose parses the deployment descriptors in the JAR file and sets up the associations and stereotypes automatically.

Associations between the various elements in the EJB—the implementation class, the home interface, the remote interface, and the primary key—are displayed graphically in the Class diagram.

Special icons are used in the Logical View of the Rose browser to display the various elements in the EJB.

There are two ways to reverse engineer EJBs:

- You can drag and drop an `ejb-jar.jar` file onto a class diagram.
- You can use the filter field on the Reverse Engineering dialog box (**Java > Reverse Engineer Java**) to reverse engineer a JAR file or to reverse engineer the `.class` or `.java` files and the `.xml` files that comprise an EJB.

When you use the Reverse Engineering dialog box to select the `.class` or `.java` files that are part of an EJB, you must also select the XML deployment descriptor (the `.xml` file) in the same reverse engineering operation. The deployment descriptor defines the associations that make up an EJB. Without the deployment descriptor, Rose J cannot recreate an accurate view of your EJB classes.

Also, in order for Rose J to resolve references to J2EE API classes, you will need to update your classpath setting with a path to the `j2ee.jar` file (for example, `d:\j2sdkee1.2.1\lib\j2ee.jar`). For information about extending the classpath within Rose, see the Classpath Help topic in the Rose J Help. For additional information and requirements, see *About Reverse Engineering* in the Rose J Help.

Generating Test Assets

This section explains how to generate the various test assets you need to test EJBs with QualityArchitect. Test assets include test scripts, datapools, stubs, and lookup tables.

Generating EJB Test Scripts

QualityArchitect provides two ways to create EJB test scripts. The method you choose depends on the processes in place on your development and testing team and on the state of the component-under-test.

The recommended way to create test scripts is to model the component-under-test in Rational Rose and generate test scripts from the Rose model. However, if you have already built or partially built a component, you can do the following:

- Reverse engineer the component into Rose and generate test scripts from the model.
- Use the EJB Session Recorder to create an XML log file and generate test scripts from the log file.

Generating Unit Test Scripts from a Rose Model

You can use the sample model that is included with the Rational Bank Account Sample Application to generate unit test scripts from Rose. For example, try generating a unit test for the `getBalance` method, which is part of the remote interface named `ExecuteTransaction`. This method is used to obtain the current balance from the specified account.

The `getBalance` method calls three methods from the session bean `ManageAccounts` in its code. The three methods are `getSavingsCustomerID`, `getSavingsBalance`, and `getCheckingBalance`. All of these methods take non-complex objects as parameters.

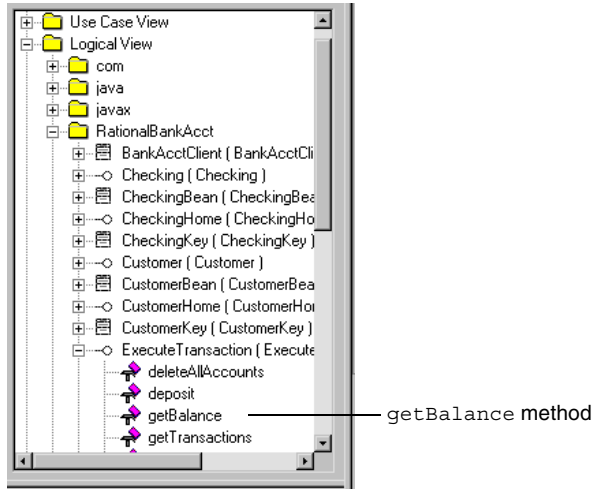
To generate the unit test:

- 1 If you haven't already done so, create a project with the Rational Administrator for maintaining your test assets.

For details, see *Adding a Project* in the Administrator Help.

- 2 Start Rose and open the `ratlbankacct.mdl` model.

- 3 In Rose, right-click the `getBalance` method and click **QualityArchitect > Select Unit Test Template**.



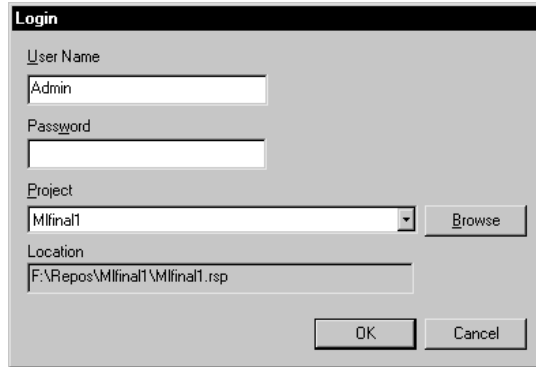
- 4 Do one of the following:

- Select the **websphere_remote.template** and click **Open** if you are working in a WebSphere environment.
- Select **weblogic_remote.template** and click **Open** if you are working in a WebLogic environment.
- Select **sunj2ee_remote.template** and click **Open** if you are working in a Sun J2EE environment.

QualityArchitect defaults to the last template chosen, so in the future you can skip this step if you plan to use the same template. For more information about templates, see *Templates* on page 7.

- 5 In Rose, right-click the `getBalance` method again and click **QualityArchitect > Generate Unit Test**.

A message appears, indicating that code generation is in progress, after which you will be prompted to log in to a Rational project.



- 6 Log in to the project and click **OK**.

Each project contains a datastore for storing test assets, such as datapools, lookup tables, and log files.

If this is the first time you have generated a script for a particular datastore, you will be prompted to select a directory in which to store your scripts.

- 7 Select a directory location and click **OK**.

QualityArchitect creates a directory hierarchy under the location you have chosen and saves the test script. The test script itself is assigned a name of the format *InterfacenameMethodname*. Thus, a script generated to test `getBalance` in the `ExecuteTransaction` interface would be named `ExecuteTransactiongetBalance.java`.

When code generation is complete, QualityArchitect reminds you to check the log file for details.

Note: QualityArchitect maintains an association between the test script directory that you have chosen and the test assets that are stored in the project's test datastore.

For information about using scenario testing to test your business logic, see *Using EJB Scenario Tests to Test Transactions* on page 57.

Generating Test Scripts with the EJB Session Recorder

If you have already built or partially built a component, you can use the EJB Session Recorder to create an XML log file and generate test scripts from the log file. For details, see *Using the EJB Session Recorder* on page 66.

Creating a Datapool for the Unit Test

A *datapool* is a set of records that you can use to drive a test script. Datapool support is included automatically in each test script generated from Rose.

The following code fragment shows the datapool name and the parameter (column) names embedded in the test script:

```
String sDPName = "ExecuteTransaction_getBalance_D";
dp.open(sDPName);
fRetVal = dp.fetch();
while (fRetVal)
{
    iDPCount = iDPCount + 1;

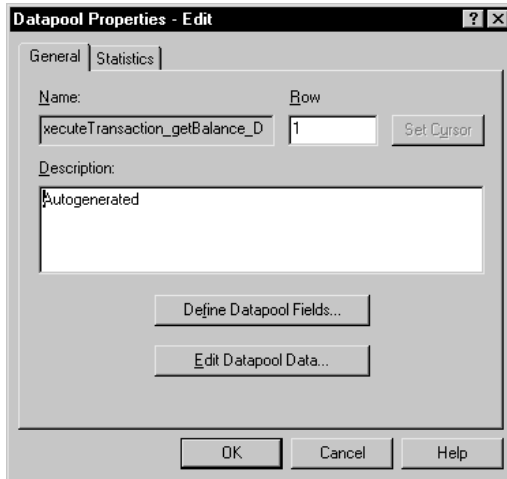
    // Retrieve values from Datapool for datatypes that we understand.
    accountID = dp.value("accountID").longValue();
    acctType = dp.value("acctType").toString();
    ExpectedReturn = dp.value("expectedReturn").getBigDecimal();
    sExpectedException = dp.value("expectedException").toString();
}
```

_____ Datapool name

|
Column names are displayed here in boldface.

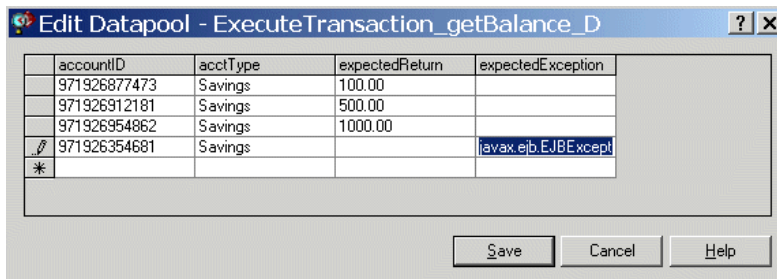
To create a datapool that you can use to test the `getBalance` method:

- 1 In the Rose browser, right-click the `getBalance` method and click **QualityArchitect > Create Datapool**.



QualityArchitect auto-generates a datapool named `ExecuteTransaction_getBalance_D`, using the parameters in the `getBalance` method (`accountID` and `acctType`) for the column names. It also creates a column for the expected return value of the method and the expected exception. The expected exception column should only be populated for rows containing input that elicit an exception from the method-under-test.

- 2 Click **Edit Datapool Data** and populate the datapool with several rows using the generated account numbers (IDs). Specify `Savings` as the account type, and enter the actual account balance in the `expectedReturn` column.



For more information about datapools, see *Datapools* on page 21, the *Test Script Services for Java* manual, and the Help for Rational TestManager.

Generating Stubs for the Unit Test

With QualityArchitect, you can create stubs for session beans and test any component that calls that session bean.

When you generate stubs for session beans, QualityArchitect creates Java source files for the home interface, the remote interface, and the implementation class itself. After generating stubs, you can deploy them on your application server, where they replace the real session beans.

Templates used for generating EJB session stubs include:

- Session_Bean.template
- Session_Home.template
- Session_Remote.template
- MethodBodyWithoutLookUp.template
- MethodBodyWithoutExceptions.template
- MethodBodyWithoutReturnValue.template
- MethodBody.template

Because the `getBalance` method calls methods in the `ManageAccounts` EJB, you can either generate a stub or run the tests directly against the actual `ManageAccounts` EJB.

To generate the stubs:

- 1 Right-click the **ManageAccountsBean** class in the Rose browser and click **QualityArchitect > Generate Stub**.
- 2 Select a directory for storing the stubs.

QualityArchitect generates stubs for:

- The remote interface, `ManageAccounts.java`
- The home interface, `ManageAccountsHome.java`
- The implementation class, `ManageAccountsBean.java`

Note: Stubs must be deployed on the same computer as the test script.

For a high-level overview of stubs, see *Stubs* on page 16.

Creating Lookup Tables

After you generate the stubs, create a lookup table for each method in the stub called by the method-under-test. In this case, you need to create two lookup tables because `ExecuteTransaction.getBalance`, when called against a `Savings` account, calls two methods in the `ManageAccounts` bean—`getSavingsBalance` and `getCustomerID`.

Lookup tables are based on Rational datapool technology. Whereas a datapool is used to test inputs and expected behavior, a lookup table is used with stubs to simulate the behavior of an actual component.

To create a lookup table for the `getSavingsBalance` method in the `ManageAccounts` bean:

- 1 From the Rose browser, select the `getSavingsBalance` method in the `ManageAccounts` bean.
- 2 Click **Tools > QualityArchitect > Stubs > Create look-up table**.
- 3 Populate the table with real data—for example:

Account ID	Expected Return	Expected Exception
<i>Generated ID 1</i>	<i>account balance</i>	
<i>incorrect account ID</i>		<code>java.rmi.RemoteException</code>

- 4 Repeat these steps for `getCustomerID`.

At this point, you should be finished creating your tests assets, and you can start setting up your development environment in order to run the tests.

Importing Test Assets into VAJ

After you generate your test assets, you must add them to a project in your IDE, where you can run the tests.

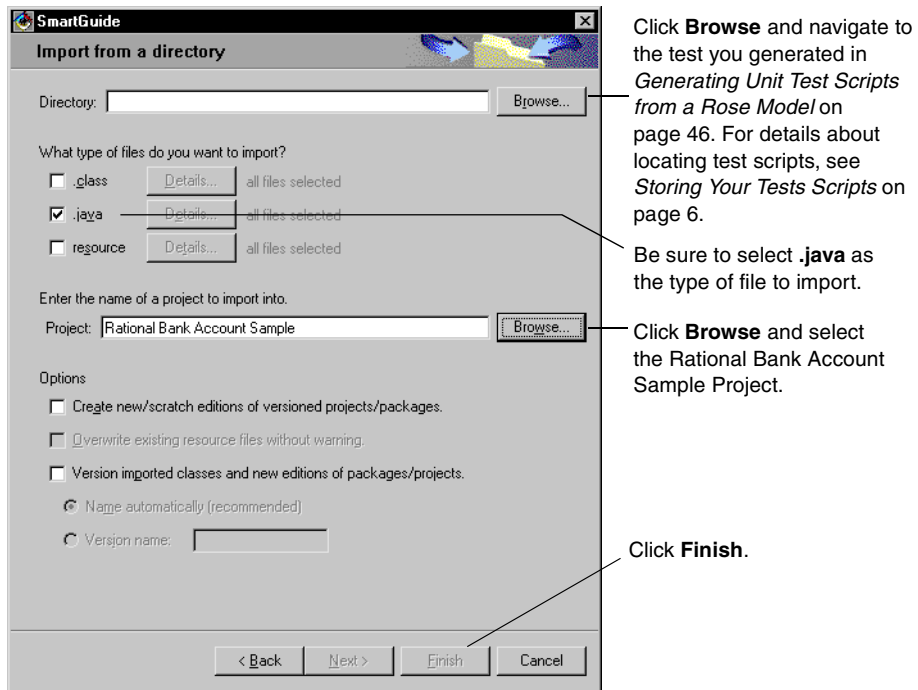
If you are using IBM Visual Age for Java (VAJ), this means importing the test into the VAJ repository.

Note: For Visual Cafe, you simply add the file to an existing project or create a new project.

Importing Unit Tests Into VAJ

To import the unit test that you generated into the RationalBankAccount project in VAJ:

- 1 Start VAJ.
- 2 Within the VAJ Workbench window, click the **Projects** tab.
- 3 Right-click anywhere in the window and click **Import**.
- 4 When the SmartGuide dialog box appears, select **Directory** as the import source and click **Next**.



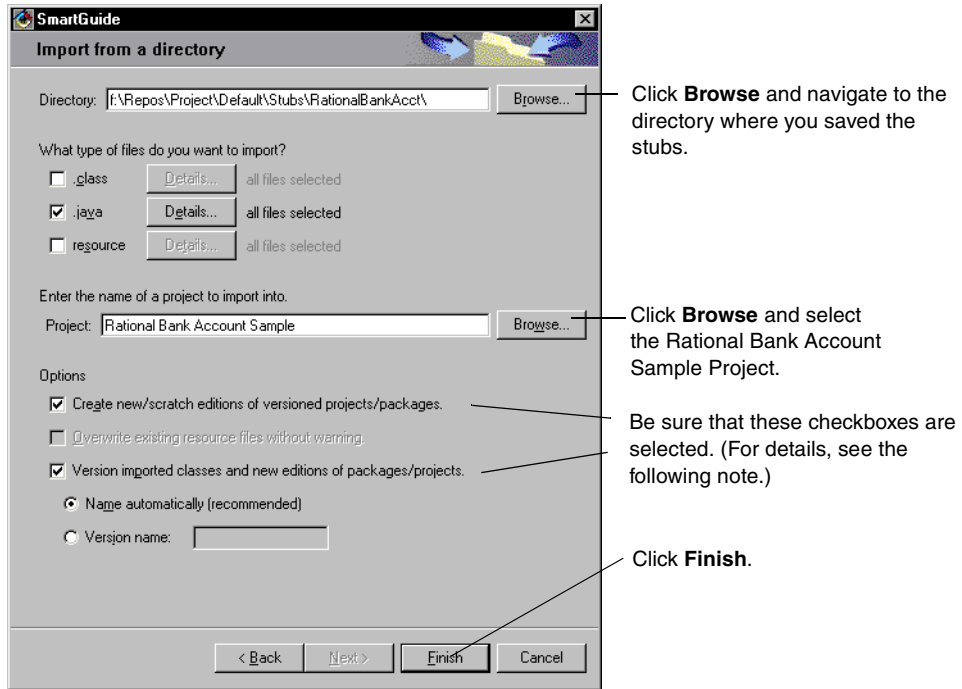
Note: When you import a script into VAJ, the script is compiled automatically.

Importing Stubs into VAJ

To import the stubs that you generated into the RationalBankAccount project in the VAJ repository:

- 1 Start VAJ if it is not already started.
- 2 Within the VAJ Workbench window, click the **Projects** tab.
- 3 Right-click anywhere in the window and click **Import**.

- 4 When the SmartGuide dialog box appears, select **Directory** as the import source and click **Next**.



Note: Be sure to select the following check boxes: **Create new/scratch editions of versioned projects/packages** and **Version imported classes and new editions of versioned projects/packages**. Doing so lets you roll back to a known version of the software and thus switch between a stubbed test environment and a non-stubbed test environment.

To roll back to a previous version:

- 1 Shut down the EJB servers.
- 2 Right-click on the RationalBankAcct package and click **Replace with > Another Edition** menu.

Executing Test Scripts

This section describes how to execute your test scripts from VAJ, Visual Cafe, and Rational TestManager.

Note: Before you can execute a test script, you may need to edit the `hostName` and `portNumber` variables for your particular, remote application server, as shown below. These variables are declared in the `getInitialContext()` method in your script and in the template used to generate the script. By default, `localhost` is used if your application server is installed locally.

```
private InitialContext getInitialContext() throws NamingException {
    InitialContext initialContext= null;
    Properties p = new Properties();
    StringcontextFactory=
"com.ibm.ejs.ns.jndi.CNInitialContextFactory";
    Stringurl= "IIOP:///";
    String portNumber = null;
    String hostName = "localhost";
```

Executing Test Scripts from VAJ

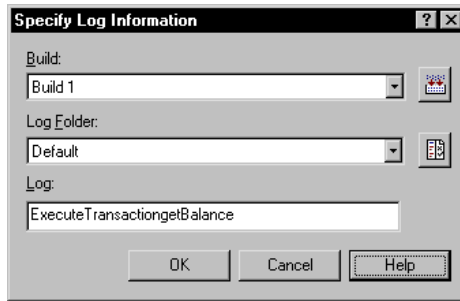
To execute the test script from VAJ:

- 1 Start the EJB servers.

For details about starting the EJB servers, see *Configuring VAJ to Use the New Sample Database* on page 35.

- 2 Right-click on the generated test script `ExecuteTransactiongetBalance` in the VAJ Project window and click **Run > Check Classpath**.
- 3 Click **Compute Now** to add the path of the generated test script to the ClassPath.
- 4 If you plan to use the WebSphere Test Environment, click the **Edit** button to the right of the Project path and make sure the IBM WebSphere Test Environment package is checked. Click **OK**.
- 5 Click **OK** to accept the generated class path.
- 6 Right-click on the generated test script `ExecuteTransactiongetBalance` in the VAJ Project window and click **Run > Run main**.
- 7 If necessary, log in to a project and click **OK**.

The Specify Log Information dialog box appears:



- 8 Accept the defaults in the Specify Log Information dialog box and click **OK**.

Click **Help** for more information about the Specify Log Information dialog box.

- 9 View the test results in the Rational Test Log window.

The log should contain entries from the stub, indicating the parameters received and the values returned.

For more information about the Test Log window, click **Help**.

Executing Test Scripts from Visual Cafe

After you have added the required JAR files (see *Requirements for EJB Testing* on page 29), executing test scripts from Visual Cafe is simply a matter of adding the test scripts to an existing or new project and clicking **Project > Execute**.

Executing Test Scripts from TestManager

Essentially, if you can run your client application from a command prompt, you should be able to execute your test scripts from TestManager. When running a Java test script, TestManager calls whatever Java compiler you are using to compile the test script. For a list of requirements, see *Rational TestManager Playback Requirements* on page 31.

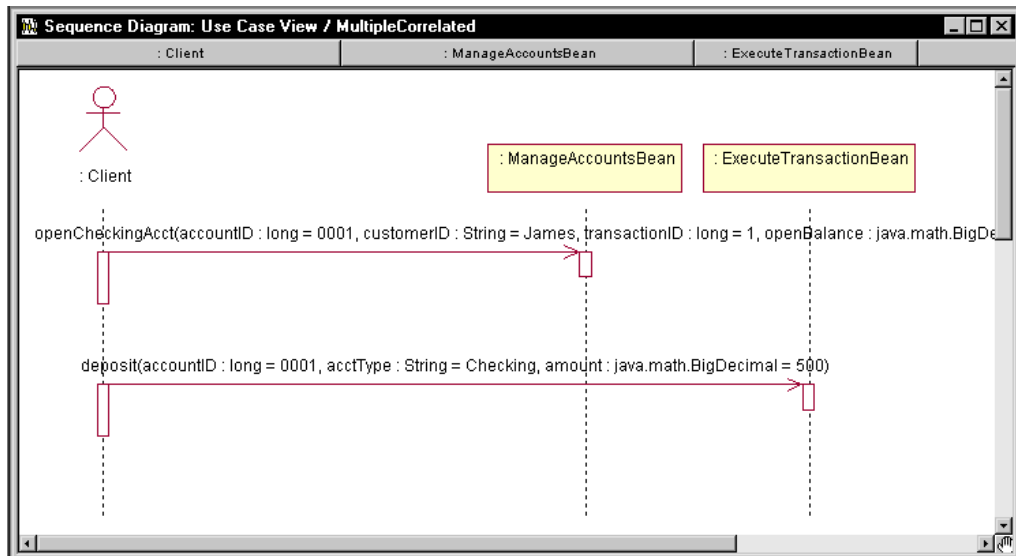
To execute an RQA Java test script in TestManager:

- 1 Start TestManager.
- 2 Click **File > Run Test Script > RQA Java Test Scripts**.
- 3 Select the script and click **Open**.
- 4 From the Run Script dialog box, click **OK**.

For additional information about other ways to run test scripts from TestManager, see the *Using Rational TestManager* manual or the TestManager Help.

Using EJB Scenario Tests to Test Transactions

Scenario tests use Rose interaction diagrams to test transactions. To try out this feature, you can generate a scenario test for one of the sequence diagrams that are included with the Rational Bank Account sample application. For example, the following figure shows the Multiple Correlated sequence diagram that is included with the sample application



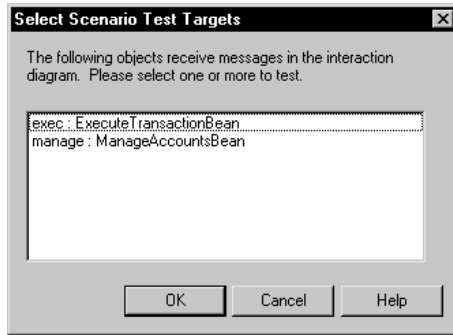
When you generate a scenario test, QualityArchitect prompts you to insert a verification point for each message in your interaction diagram. To add a database verification point, you will need a database, and you will need to know the fully qualified path of your JDBC driver and the JDBC URL, which also includes the name of the database. For information about setting up the database for the sample application, see *The Rational Bank Account Sample Application* on page 32.

To generate a scenario test:

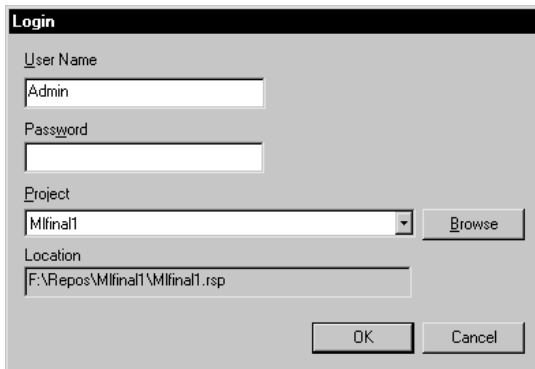
- 1 Right-click in an active interaction diagram and click **QualityArchitect > Select Scenario Test Template**.
- 2 Expand either the WebSphere folder, the WebLogic folder, or the Sun J2EE folder, and select the appropriate scenario template for your environment.
- 3 Click **Open**.
- 4 Right-click in the interaction diagram and click **QualityArchitect > Generate Scenario Test**.

- 5 In the Select Scenario Test Targets dialog box, select the scenario test targets—that is, the receiver objects (where the arrows are pointing).

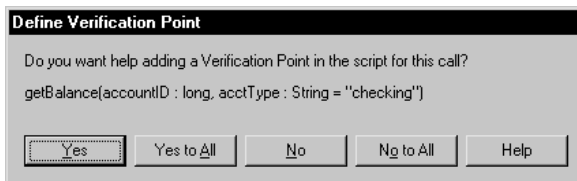
You can select one or more objects to test.



- 6 Click **OK**.
- 7 If prompted, log in to a project and click **OK**.



- 8 In the Define Verification Points dialog box, click **Yes** to add a verification point for the first message in the diagram—`getBalance`.



- 9 In the Select Verification Point Type dialog box, select a verification point type and click **OK**.

For this release, select the database verification point type—`DatabaseVP`.

- 10 At this point, the Query Builder wizard starts. Use the Query Builder to connect to the database and to define a Select statement that can be used to query the database. For more information about the Query Builder, see *Using the Java Query Builder to Add Verification Points* on page 59.
- 11 Repeat Steps 8 and 9 for each message in the diagram.
- 12 After the last message, you are prompted to add a verification point at the end of the scenario. Click **Yes**.
- 13 Select a directory to store the tests in and click **OK**.

Note: You can also start the scenario test generator by right-clicking on a diagram in the Rose browser and then clicking **QualityArchitect > Generate Scenario Test**. However, when you generate code by right-clicking on a diagram name in the Rose browser, code generation always proceeds using the active diagram—the one that is open in a window—not necessarily the diagram that you clicked on.

Using the Java Query Builder to Add Verification Points

The QualityArchitect Query Builder is a tool that helps you connect to and interact with databases for the purpose of defining database verification points. The Query Builder uses a wizard-like interface that lets you build a query one step at a time. Once built, the query is used with a database verification point.

To understand more about the Query Builder, consider what happens if you generate a scenario test for the Multiple Correlated sequence diagram in the sample application. (See *Using EJB Scenario Tests to Test Transactions* on page 57.) This sequence diagram contains the following messages:

- `openCheckingAcct`
- `deposit`

In testing this scenario, you could choose to create a verification point for each message. The first verification point could be used to verify that an account was opened and that the opening balance is as expected. The second verification point could be used to verify the results of a deposit to this account.

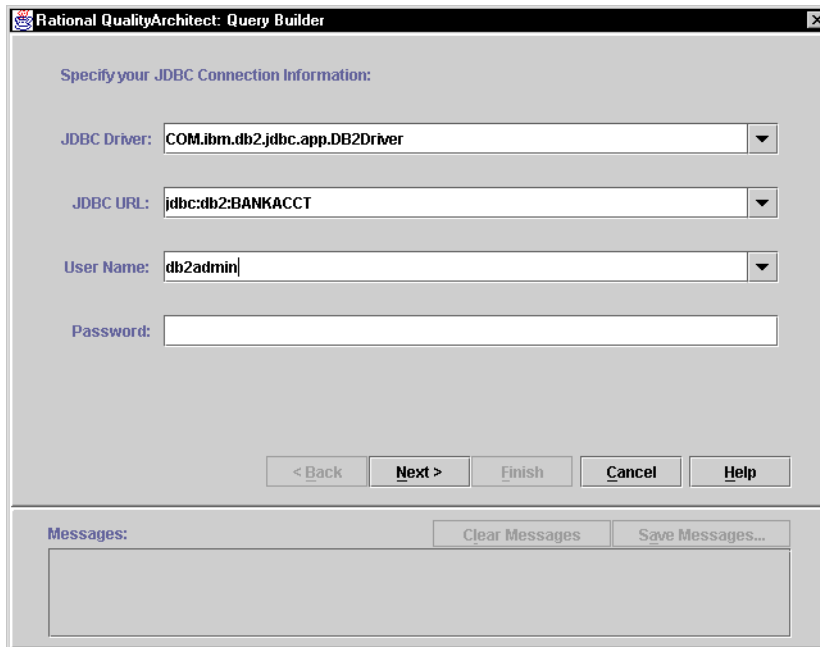
To design the query, you must do the following:

- Connect to a database via JDBC.
- Design a SQL statement.
- View the SQL query results.
- Verify the specified SQL query settings.

Connecting to the Database via JDBC

The first step involved in building a custom SQL query is to connect to the database and entering the required JDBC information.

When the Query Builder starts, specify the JDBC Driver, the JDBC URL, a user name, and password, and then click **Next**.



Rational QualityArchitect: Query Builder

Specify your JDBC Connection Information:

JDBC Driver:

JDBC URL:

User Name:

Password:

< Back Next > Finish Cancel Help

Messages: Clear Messages Save Messages...

JDBC Driver

The JDBC Driver text is the fully qualified class name of the Java JDBC driver class that was written for the specific DBMS that you would like to connect to. The following table lists some examples of JDBC Driver text for some common environments:

Environment	JDBC Driver Text
WebSphere/DB2	COM.ibm.db2.jdbc.app.DB2Driver
WebLogic/SQL Server	weblogic.jdbc.mssqlserver4.Driver
Any ODBC data source	sun.jdbc.odbc.JdbcOdbcDriver

JDBC URL

The JDBC URL provides a way of identifying a database to the driver. The standard syntax for JDBC URLs is as follows:

```
jdbc:<subprotocol>:<subname>
```

The three parts of a JDBC URL are broken down as follows:

Part of URL	Description
jdbc	The jdbc protocol. The protocol is always jdbc.
subprotocol	Contains the name of the driver or the name of a database connectivity mechanism, which may be supported by one or more drivers. A prominent example of a subprotocol name is "odbc", which has been reserved for URLs that specify ODBC-style data source names.
subname	Used to identify the database. The subname can vary, depending on the subprotocol, and it can have a subsubname with any internal syntax that the driver writer chooses.

The following table lists some examples of JDBC URLs for some common environments:

Environment	JDBC Driver Text
WebSphere/DB2	jdbc:db2:BANKACCT
WebLogic/SQL Server	jdbc:weblogic:mssqlserver4:CQSMV
Any ODBC data source	jdbc:odbc:nwind

User Name

The User Name text is the name of the user who has access rights to the specified database.

Password

The Password text is the password that is associated with the database user.

Designing a Custom Query Statement

After you connect to the database, you can design the custom SQL query statement that will be used to retrieve specific data from the database.

For example, in testing `openCheckingAcct`, you could construct a `Select` statement that returns an account ID, a customer ID, and a balance from the `Checking` table.

If you are familiar with SQL syntax and are familiar with the schema of the database that you are connecting to, you can simply enter your custom SQL query statement in the SQL Text box.

Once you are satisfied with your custom SQL query statement, click **Next** to apply the SQL query to the connected database.

Using the Query Design Wizard

If you are not familiar with SQL syntax or are not familiar with the schema of the database that you are connecting to, you can use the Query Design Wizard to interactively walk through the design of your custom SQL query statement. The Query Design Wizard helps you easily create complex SQL query statements by taking you step-by-step through the design process.

To use the Query Design Wizard:

- 1 Click the **Query Design Wizard** button.

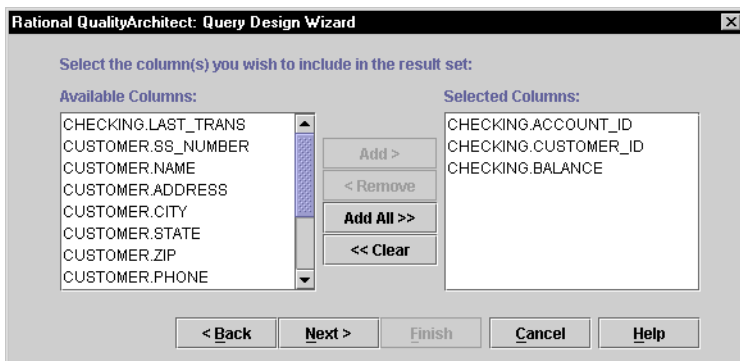


- 2 Select the tables that you want to query and click **Next**.



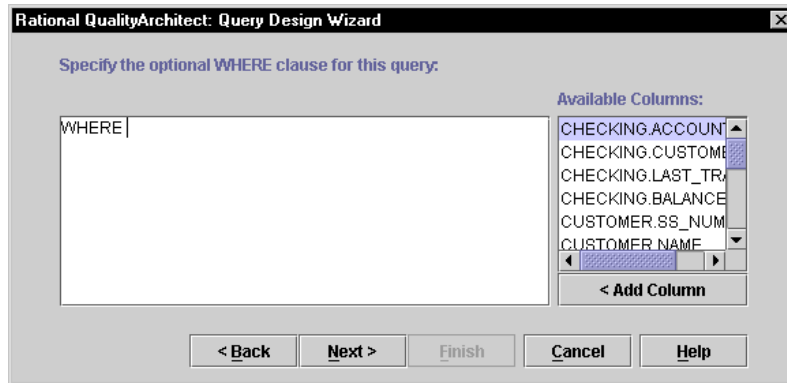
To select specific tables, click on the table names in the Available Tables list and click **Add**. To select all of the tables in the list, click **Add All**.

- 3 Select the columns to include in the result set, and then click **Next**:



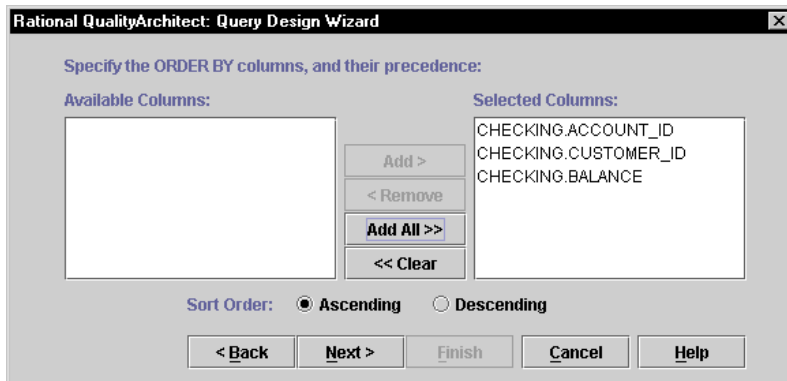
Column names are listed in *TableName.ColumnName* format so that you can easily identify which columns exist in which tables.

- 4 Enter an optional WHERE clause for the query and click **Next**.



The WHERE clause allows you to specify a search condition that will restrict the returned query results.

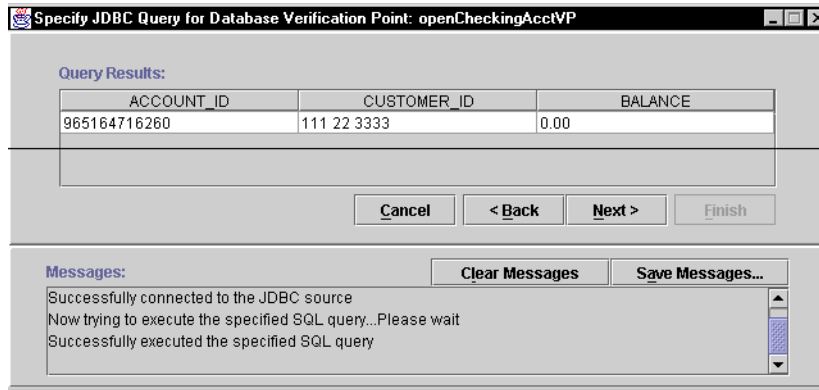
- 5 Specify the ORDER in which you want to display the columns and their precedence, and then click **Next**.



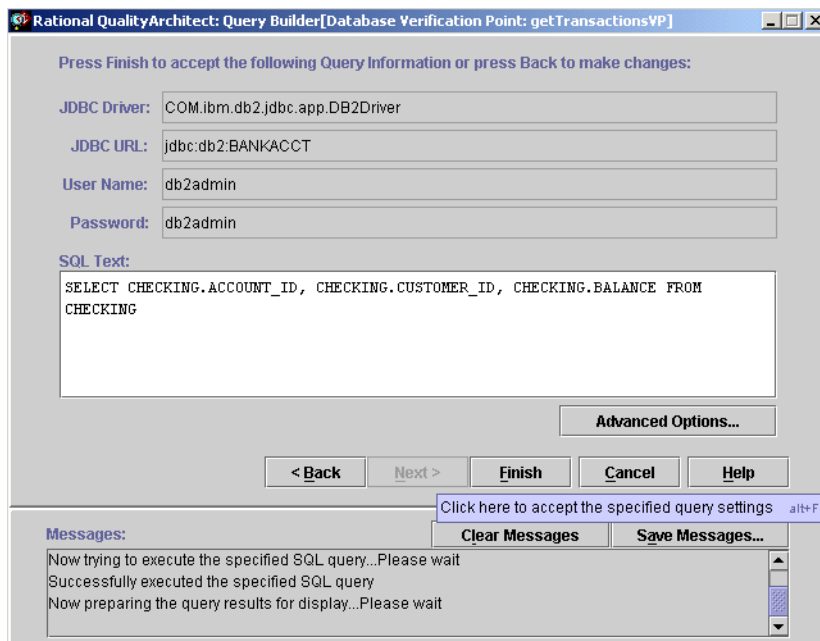
- 6 Verify the SQL statement, make any edits as needed, and click **Finish** to accept the SQL query statement and exit the Query Design Wizard.

Viewing and Verifying the SQL Query Results

When the custom SQL query has been successfully executed on the connected database, the query results are displayed in the Query Results table. The columns of the table correspond to the columns that you included in the SQL query in the SELECT clause.



- 1 If you are satisfied with the query results, click **Next**. If you need to make changes to the SQL query statement, click **Back**.
- 2 Click **Finish** to confirm the information that you entered.



See the Query Builder Help for a description of the advanced Database VP Options.

- 3 After you finish creating the query for a scenario test, the script generator asks if you want to create verification points for each additional message in the diagram. If you choose to do so, create a database verification for the `deposit` message.

After you finish entering database verification points, the scenario test script is generated. The values specified in the Query Builder are placed into a datapool, and the generated code retrieves these datapool values, allowing the verification point to be data driven.

A sample verification point is listed in the script as follows:

```
// A verification point may be automatically inserted below.
String svp1JDBCdriver = dp.value("svp1JDBCdriver").toString();
String svp1JDBCurl = dp.value("svp1JDBCurl").toString();
String svp1JDBCuser = dp.value("svp1JDBCuser").toString();
String svp1JDBCpassword = dp.value("svp1JDBCpassword").toString();
String svp1SQL = dp.value("svp1SQL").toString();
String svp1VPname = dp.value("svp1VPname").toString();
DatabaseVP vp1openCheckingAcctVP = new DatabaseVP( "svp1VPname",
svp1SQL, svp1JDBCuser, svp1JDBCpassword, svp1JDBCdriver,
svp1JDBCurl);

vp1openCheckingAcctVP.performTest(null);
```

For more information about coding verification points, see the *Rational Test Script Services for Java* manual.

Using the EJB Session Recorder

The EJB Session Recorder is a tool that lets you visually connect to and interact with EJBs. As you execute transactions against the component, interaction data is recorded and stored in an external XML file. Afterwards, you can use the XML Script Generator to generate test scripts for testing the EJB.

Before working with the EJB Session Recorder, see *Requirements for EJB Testing* on page 29 for information about any required software.

The basic steps for using the EJB Session Recorder are as follows:

- 1 Start the EJB Session Recorder.

- 2 Connect to the JNDI (Java Naming and Directory Interface) Naming Service and select a deployed EJB.

This returns the EJB's home interface.

- 3 Select a method on the home interface and enter parameter names and values for the method.

- 4 Invoke the method.

This returns the EJB's remote interface.

- 5 Select a method on the remote interface and enter parameter names and values for the method.

For example, to enter parameter names and values for the EJBs installed with QualityArchitect, you will need to examine the Method Specifications in Rose or review the EJB's methods in your IDE.

- 6 Invoke the method.
- 7 View the XML log by clicking **View > Current XML Log**.
- 8 Generate a test script from the XML log.

EJB Session Recorder Requirements

The EJB Session Recorder is dependent on the following JAR files:

Package Description	File Name	Default Installation Path
Rational Test Script Services	rational_ct.jar rttssjava.jar	C:\Program Files\Rational\Rational Test\QualityArchitect and C:\Program Files\Rational\Rational Test
XML Script Generator	scriptgen_ct.jar	C:\Program Files\Rational\Rational Test\QualityArchitect
Swing v.1.1	swingall.jar	C:\Program Files\Rational\Rational Test\QualityArchitect
Xerces XML parser v.1.2	xerces.jar	C:\Program Files\Rational\Rational Test\QualityArchitect
Java Collections v.1.1	collections.jar	C:\Program Files\Rational\Rational Test\QualityArchitect
JavaHelp v.1.1	jh.jar	C:\Program Files\Rational\Rational Test\QualityArchitect

To ensure that the EJB Session Recorder launches successfully, these JAR files must exist in the classpath. When the EJB Session Recorder is launched from the QualityArchitect Console, these JAR files are automatically added to the classpath.

The EJB Session Recorder is also dependent on the JAR files for the EJB client application and any required application server classes. Consult your application server documentation for information about the required application server classes.

The EJB Session Recorder is compatible with JDK versions 1.1x and later.

The name of the JAR file for the EJB Session Recorder itself is `ejbtestclient_ct.jar`.

Setting Up the Classpath for the EJB Session Recorder

Before you can start the EJB Session Recorder from the Rational QualityArchitect console, several required classes must be referenced in the additional classpath for the EJB Session Recorder or in the user classpath. These classes and the order in which they must be listed are as follows:

- The home and remote interface classes for the deployed EJBs. These classes can be referenced in a directory or in a JAR file.
- The required application server classes. Consult your application server documentation for information about the required application server classes.
- The path to the installed J2EE SDK. This path must appear in the `J2EE_HOME` system environment variable.

To modify the Additional Classpath field:

- 1 Click **Tools > QualityArchitect > Console** to display the RQA Console.
- 2 Click the **RQA Options** icon.
- 3 Click the **Java Options** tab.
- 4 Type the required classes in the additional classpath field.

Application Server Classes for WebSphere

The following JAR files must be added to the additional classpath if the EJB Session Recorder needs to connect to a WebSphere application server outside of the WebSphere test environment:

- `ejs.jar`
- `ibmjndi.jar`
- `ns.jar`

- ujc.jar
- ibmwebas.jar

Application Server Classes for WebLogic

The following JAR file and directory must be added to the additional classpath if the EJB Session Recorder needs to connect to a WebLogic application server:

- weblogicaux.jar
- WebLogic\classes

Setting Up the Java Runtime Environment (JRE)

To choose the JRE that the Session Recorder runs in:

- 1 Click **Tools > QualityArchitect > Console** to display the RQA Console.
- 2 Click the **RQA Options** icon.
- 3 Click the **Java Options** tab.
- 4 Specify the desired JRE executable file in the Java Runtime Environment box.

Note: In order to connect to a WebSphere application server outside of the WebSphere test environment, you must choose the IBM JRE, which is located by default in the JDK\JRE\bin directory under the WebSphere installation directory. You must also include the JDK\JRE\bin directory in your system path.

Starting the EJB Session Recorder

There are several ways to start the EJB Session Recorder:

- From the QualityArchitect console
- From Visual Cafe
- From IBM Visual Age for Java

Note: We recommend that you start the EJB Session Recorder from the console or the command line, because it is more complicated to set up and configure the EJB Session Recorder from the IDE. In addition, performance of the EJB Session Recorder is best if run outside an IDE.

Starting the EJB Session Recorder from the Console

To start the EJB Session Recorder from the console:

- 1 Set up the classpath. (See *Setting Up the Classpath for the EJB Session Recorder* on page 68.)

- 2 Click the **EJB Session Recorder** icon on the QualityArchitect console.

Starting the EJB Session Recorder from Visual Cafe

In order for the EJB Session Recorder to successfully connect to deployed EJBs in Visual Cafe, the home and remote interface classes for those EJBs must be referenced in the Visual Café classpath.

In addition, the required application server classes must be referenced in the Visual Cafe classpath. Consult your application server documentation for information about the required application server classes.

To start the EJB Session Recorder from Visual Cafe:

- 1 Create a new, empty project in Visual Cafe.
- 2 After the new project is created, click **Project > Options** from the main menu.
- 3 Click the **Project** tab and enter the following text in the **Main Class** text box:
`com.rational.test.ejbclient.EJBSessionRecorder`
- 4 Click the **Directories** tab and verify that **Input class files** is selected in the **Show directories for** box.
- 5 Click the **New** button (the button furthest to the left) in the Directories area.
- 6 Click the **file** button.
- 7 In the File Open dialog box, navigate to the installation directory for QualityArchitect (Rational\Rational Test\QualityArchitect), select **swingall.jar** and click **Open**.

The imported JAR file appears in the **Directories** list.

- 8 Repeat the JAR import instructions for the rest of the required JAR files:
 - Rational_ct.jar
 - Xerces.jar
 - collections.jar
 - Jh.jar
 - Scriptgen_ct.jar
 - Ejbtestclient_ct.jar
 - rttssjava.jar (located under Rational Test, not QualityArchitect)
- 9 Click **Project > Execute** from the main menu to launch the EJB Session Recorder.

Passing in Command Line Parameters (optional)

As an added feature, the JNDI Provider URL and Initial Context Factory can be passed in as command line parameters for the EJB Session Recorder.

To specify these parameters for the EJB Session Recorder:

- 1 Click **Project > Options** from the main menu.
- 2 When the Project Options dialog box opens, enter the command line parameters in the Program Arguments text box.

An example for Visual Cafe would be:

```
provider=t3://localhost:7001  
initialcontext=weblogic.jndi.WLInitialContextFactory
```

Note: The JNDI Provider URL and the Initial Context Factory can also be set in the Options dialog box, which can be accessed from the Rational QualityArchitect console. (To access the console, click **Tools > QualityArchitect > Console**.)

Starting the EJB Session Recorder from Visual Age for Java 3.0x

Before you start setting up the EJB Session Recorder in VAJ v.3.0x, make sure that the IBM EJB Development Environment feature and IBM WebSphere Test Environment feature have been added. You will know that they have been added because you will see an EJB tab in the Workbench window. For details, see *Adding EJB Support* on page 35.

Creating a New Project

From the VAJ Workbench window, create a new project.

After you create the project, you will see the name of the new project in the current projects list.

Importing Swing Into the Project

- 1 Right click on the name of the project that you just created and click **Import**.
- 2 Make sure the Jar file option is selected and click **Next**.
- 3 Under **What types of files do you want to import**, make sure that the **class** and **resource** options are checked.
- 4 Under **Options**, make sure **Create new/scratch editions of versioned projects/packages** is checked.
- 5 Click the **Browse** button and navigate to the installation directory for QualityArchitect (Rational\Rational Test\QualityArchitect).

6 Select `swingall.jar` and click **Open** in the File Open dialog box.

7 Click **Finish**.

All of the class files from the JAR file will then be imported into the VAJ repository.

After **swingall.jar** is imported, an importing problems dialog box is displayed. The dialog box essentially states that the following Macintosh Swing class cannot be changed:

```
com.sun.java.swing.plaf.mac.MacBorders.class
```

You can ignore these problems because they will not affect the EJB Session Recorder.

8 Click **OK** to close the problems dialog box.

The Modify Palette dialog box now appears. This dialog box lets you specify which Swing classes you want to add to the visual palette.

9 Click **Cancel** to close this dialog box.

Importing JavaHelp

The next JAR file to import is JavaHelp (`jh.jar`). Follow the steps in *Importing Swing Into the Project* on page 71 to import JavaHelp into the project. Be sure to select `jh.jar`.

When you click **Finish**, all of the class files from the JAR file are imported into the VAJ repository.

Note: A status message appears stating that seven problems were found. These problems are caused because VAJ 3.0x uses JDK v. 1.1.7, and the minimum JDK required for JavaHelp is JDK v.1.1.8. The following is a list of the classes that are referenced by JavaHelp but are missing from JDK 1.1.7:

- `java.awt.print.PageFormat`
- `java.awt.print.Printable`
- `java.awt.print.PrinterJob`
- `java.security.PrivelegedExceptionAction`
- `java.security.Permission`

These missing classes will not affect the viewing of JavaHelp from the EJB Session Recorder in VAJ 3.0x.

Importing Xerces

The next JAR file to import is `xerces.jar`. Follow the steps in *Importing Swing Into the Project* on page 71. Be sure to select `xerces.jar`.

When you click **Finish**, all of the class files from the JAR file are imported into the VAJ repository.

Note: A problems dialog box appears stating that **One or more classes were imported into pre-existing packages in other projects**. Three of the packages found in `xerces.jar` are shared by the IBM XML Parser. Ignore these problems because they will not affect the EJB Session Recorder. Click **OK** to close this dialog box.

Importing the Remaining Jar Files from the QualityArchitect Directory

Follow the steps in *Importing Swing Into the Project* on page 71 to import the following JAR files from the QualityArchitect directory:

- `collections.jar`
- `scriptgen_ct.jar`
- `rational_ct.jar`
- `ejbtestclient_ct.jar`

Importing rtssjava.jar

The next JAR file to import is `rtssjava.jar`, which is part of Rational Test Services. Follow the steps in *Importing Swing Into the Project* on page 71. Be sure to select `rtssjava.jar`, which is located in the Rational Test directory, one directory above the QualityArchitect directory.

Importing Deployed EJBs

In order for the EJB Session Recorder to successfully connect to deployed EJBs in VAJ, you must import the home and remote interface classes for those EJBs into VAJ.

Checking the Classpath and Launching the EJB Session Recorder

- 1 Expand the package named `com.rational.test.ejbclient` under the project you created by clicking on the + box next to the package name.
- 2 Right-click on the `EJBSessionRecorder` class and click **Run > Check Class Path**.
- 3 Click the **Compute Now** button to fill in all of the package dependencies for the EJB Session Recorder.

- 4 If you plan to use the WebSphere Test Environment, click the **Edit** button to the right of the Project path and make sure that the IBM WebSphere Test Environment package is checked. Then, click **OK**.
- 5 Click **OK**.
- 6 Right-click on the EJBSessionRecorder class again and click **Run > Run main** to launch the EJB Session Recorder.

Passing in Command Line Parameters (optional)

As an added feature, the JNDI Provider URL and Initial Context Factory can be passed in as command line parameters for the EJB Session Recorder.

To specify these parameters for the EJB Session Recorder:

- 1 Right-click the EJBSessionRecorder class and click **Run > Check Class Path**.
- 2 When the project properties dialog box opens, click the **Program** tab.
- 3 Enter the command line parameters in the Command line arguments text box.

An example for VAJ would be:

```
providerurl=iiop:///
contextfactory=com.ibm.ejs.ns.jndi.CNInitialContextFactory
```

Starting the EJB Session Recorder from Visual Age for Java 3.5

Before you start setting up the EJB Session Recorder in VAJ v.3.5, make sure that the IBM EJB Development Environment feature and IBM WebSphere Test Environment feature have been added. You will know that they have been added because you will see an EJB tab in the Workbench window. For details, see *Adding EJB Support* on page 35.

To start the EJB Session Recorder from VAJ 3.5:

- 1 Follow the steps described for VAJ 3.0x in *Starting the EJB Session Recorder from Visual Age for Java 3.0x* on page 71 and *Creating a New Project* on page 71.
- 2 Because Swing is included in VAJ3.5, you can skip the step *Importing Swing Into the Project* on page 71.
- 3 Follow the steps described for VAJ 3.0x in *Importing Xerces* on page 73.
- 4 Follow the steps described for VAJ 3.0x in *Importing the Remaining Jar Files from the QualityArchitect Directory* on page 73. Be sure to import the following JAR files:
 - jh.jar
 - collections.jar

- scriptgen_ct.jar
 - rational_ct.jar
 - ejbtestclient_ct.jar
- 5 Follow the steps described for VAJ 3.0x in *Importing rttssjava.jar* on page 73.
 - 6 Expand the package named com.rational.test.ejbclient under the project you created by clicking on the + box next to the package name.
 - 7 Right-click on the EJBSessionRecorder class and click **Run > Check Class Path**.
 - 8 Click the **Compute Now** button to fill in all of the package dependencies for the EJB Session Recorder.
 - 9 If you are going to run the EJB Session Recorder under the IBM WebSphere Test Environment, you must click the **Edit** button and check **IBM WebSphere Test Environment**.
 - 10 Click **OK**.
 - 11 Right-click on the EJBSessionRecorder class again and click **Run > Run main** to launch the EJB Session Recorder.
 - 12 Optionally, follow the steps described for VAJ 3.0x in *Passing in Command Line Parameters (optional)* on page 74.

Using the EJB Session Recorder with the Sample Application

You can use the EJBs that have been installed with the sample application to try out the EJB Session Recorder.

In this section, you will:

- 1 Start a recording session.
- 2 Connect to an EJB.
- 3 Interact with the home interface.
- 4 Interact with the remote interface.
- 5 Insert a verification point.
- 6 View the XML log.
- 7 Generate a test script.

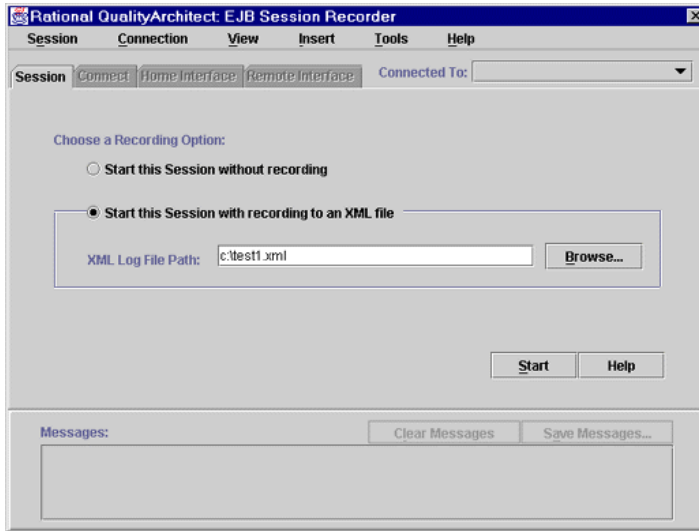
Before starting the EJB Session Recorder, be sure to start your Application Server.

Starting a Recording Session

To start a recording session:

- 1 Start the EJB Session Recorder, using one of the methods listed in *Starting the EJB Session Recorder* on page 69.

When you start the EJB Session Recorder, the first screen that appears is the Session panel.



- 2 Click **Start this Session with Recording to an XML** file.
- 3 Specify a path in which to store the XML log file.
- 4 Click **Start**.

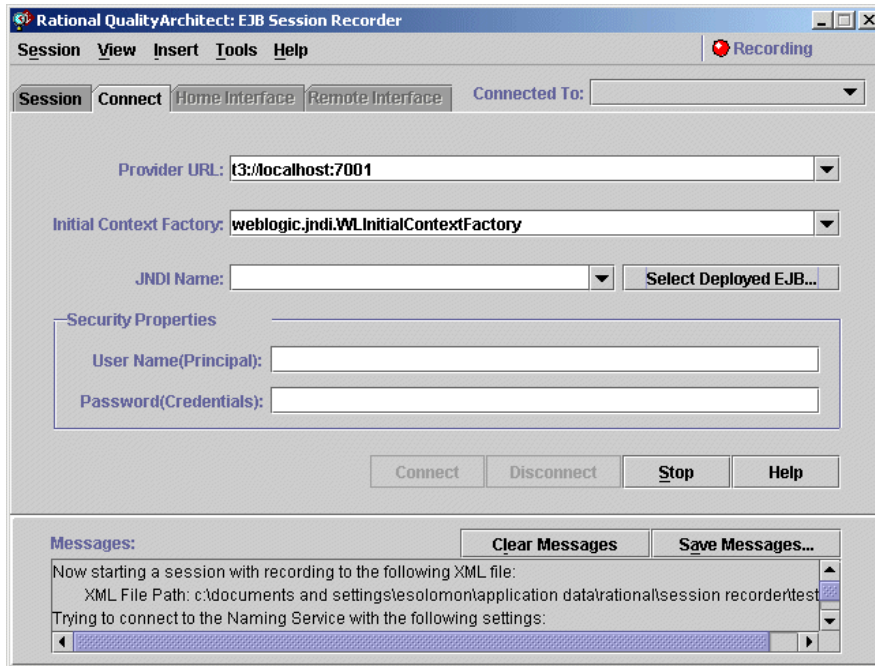
The Connect Panel appears.

Connecting to an EJB

Use the Connect Panel to enter the settings required to connect to JNDI, to find the EJB stored in the Naming Service, and to return the home interface.

Application servers, such as IBM WebSphere and BEA WebLogic, may utilize a completely different Naming Service architecture. In most cases, you should be able to accept the default settings for your application server.

A sample Connect Panel appears in the following figure:



To connect to a deployed EJB and return its home interface:

1 Specify the **Provider URL**.

The **Provider URL** is used to specify the location of the server that is used to provide the Naming Service.

- The default URL for the WebSphere Naming Service server is `iiop://`.
- The default URL for the WebLogic Naming Service server is `t3://localhost:7001`.
- The default URL for the Sun J2EE Naming Service server is `iiop://localhost:1050`.

The general format for the **Provider URL** is:

`service://host/port/`

where

- `service` refers to the name given to the Naming Service for the Application Server.
- `host` is the host machine that the Naming Service is running on.
- `port` refers to the specific port on the host machine that the Naming Service is listening to for requests.

2 Specify the **Initial Context Factory** for your application server.

To access a JNDI service provider, you need to create an initial context and pass it a context factory, which creates a context for a particular provider. The *context factory* is essentially a Java object that contains the JNDI information required by the Naming Service. Once the context is created, it provides client access to the Naming Service.

The default context factory for WebSphere is:

`com.ibm.ejs.ns.jndi.CNInitialContextFactory`

The default context factory for WebLogic is:

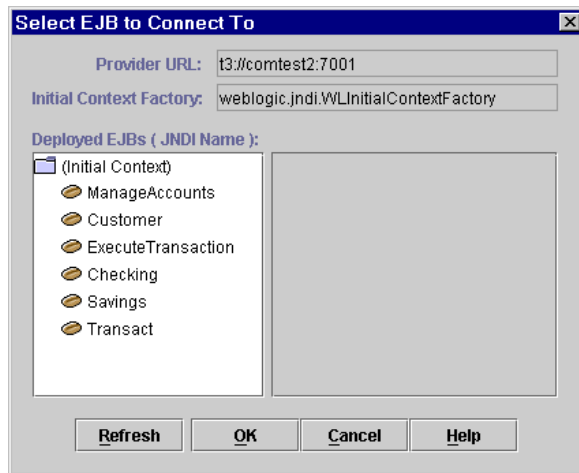
`weblogic.jndi.WLInitialContextFactory`

The default context factory for the Sun J2EE reference platform is:

`com.sun.jndi.cosnaming.CNCTXFactory`

3 Click the **Select Deployed EJB** button.

This retrieves the list of deployed EJBs from the application server. In this case, select the `ExecuteTransaction` bean and click **OK**.



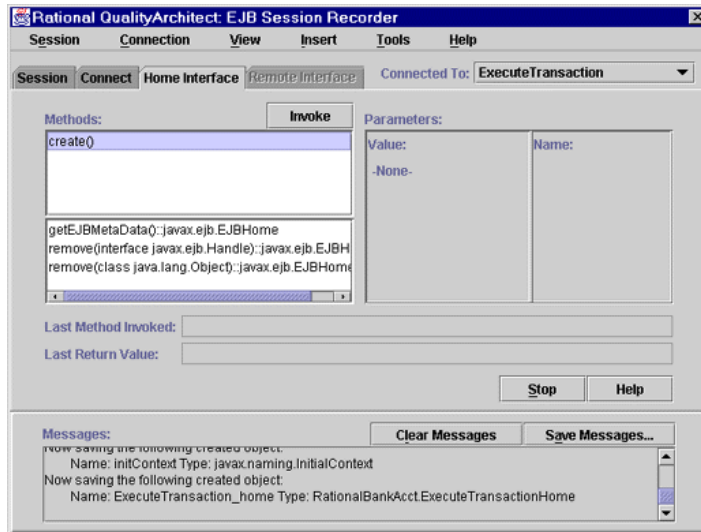
4 Click **Connect**.

This connects you to the JNDI services, which returns the deployed bean's home interface and displays the Home Interface panel.

Note: If there is a problem connecting to JNDI or retrieving the home interface, view the error information that appears in the Messages panel.

Interacting with the Home Interface

The Home Interface panel displays the public methods for the EJB's home interface.



The Home Interface panel displays methods, parameters, and messages.

- The Methods list displays all of the public methods in the EJB's home interface class.
- The Parameters pane displays all the required parameters for the selected home interface method.
- The Messages pane is used to display status information for invoked methods.

Invoking a Method on the Home Interface

To invoke a method on the home interface:

- 1 Verify that the `create()` method is selected in the **Methods** list.

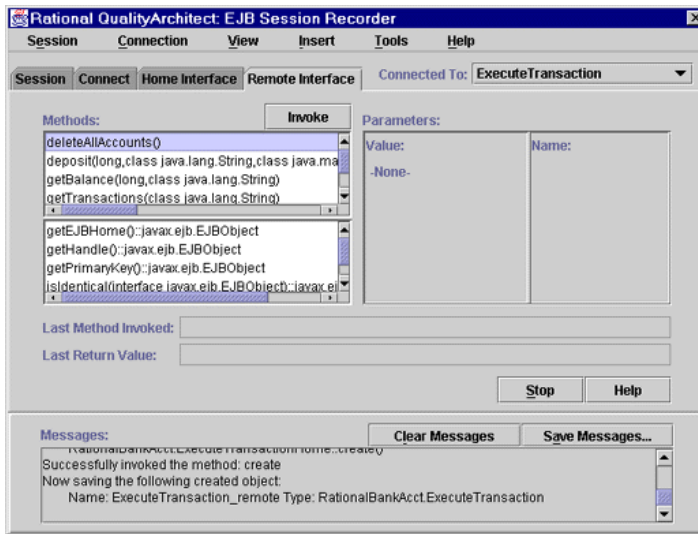
There are no required parameters for the `create()` method.

- 2 Click **Invoke**.

When you invoke the `create()` method on the home interface, the remote interface is returned and the Remote Interface panel appears.

Interacting with the Remote Interface

This panel is very similar to the Home Interface panel in that it displays methods, parameters, and messages.



- The Methods list contains all of the public methods for the EJB's remote interface. The declared methods are displayed in the top section, and the inherited methods are displayed in the bottom section.
- When you select a method, its required parameters are displayed in the Parameters pane.
- The Messages pane is used to display status information for invoked methods.

To invoke a method on the remote interface, select the method from the Methods list. If there are any parameters required to invoke the method, enter the desired values for the parameters in the Parameters pane. Then, click the **Invoke** button to execute the method.

Note: To invoke methods for the sample application's remote interface, you will need to know the arguments for each method. One way to obtain this information is to examine one of the supplied Rose models.

Invoking the openAccount() Method

For example, to invoke the `openAccount()` method on the `ExecuteTransaction` remote interface:

- 1 Click the `openAccount()` method from the **Methods** list.
- 2 Enter values for the required parameters—for example:

Values	Names
123 45 6789	ssNumber
Theadore Cleaver	name
1959 Wright Way	address
Burbank	city
California	state
98765	zip
512 399-5678	phone

Optionally, you can also enter names for these values. Names are necessary only if you plan to reuse parameters during the session.

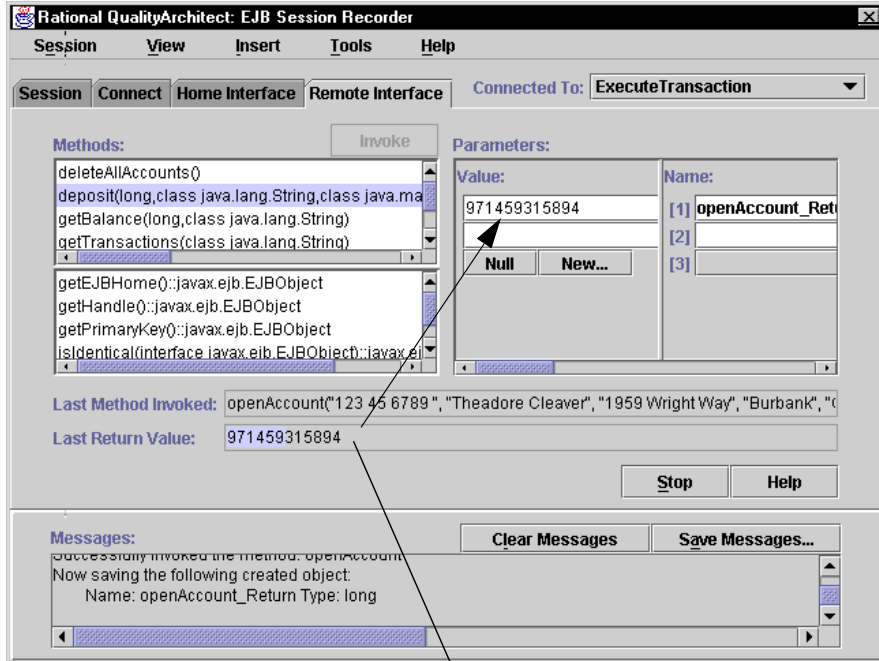
- 3 Click **Invoke**.

This operation returns an account ID in the **Last Return Value** box. The return value for `openAccount` is stored in a named variable called `openAccount_Return`.

Invoking the deposit() Method

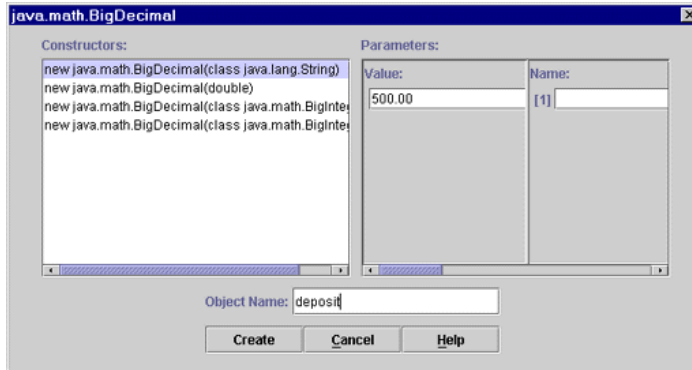
To invoke the `deposit()` method:

- 1 Click the `deposit()` method from the **Methods** list.
- 2 Click on the value displayed in the **Last Return Value** box. Drag and drop the value on to the first **Parameter Value**.



Drag and drop the **Last Return Value** on to the first **Parameter Value**.

- 3 Type **Savings** for the second **Parameter Value** and then type **accountType** in the **Name** box.
- 4 Click **New**.
- 5 In the dialog box that appears, verify that the first constructor is selected and then type a deposit amount, such as **\$500**, in the first **Parameter Value** box.



- 6 Optionally, type a name for the created object in the Object Name box.

The Object Name lets you assign a name to the created object. Named objects are stored in the program cache and can be reused in other methods.

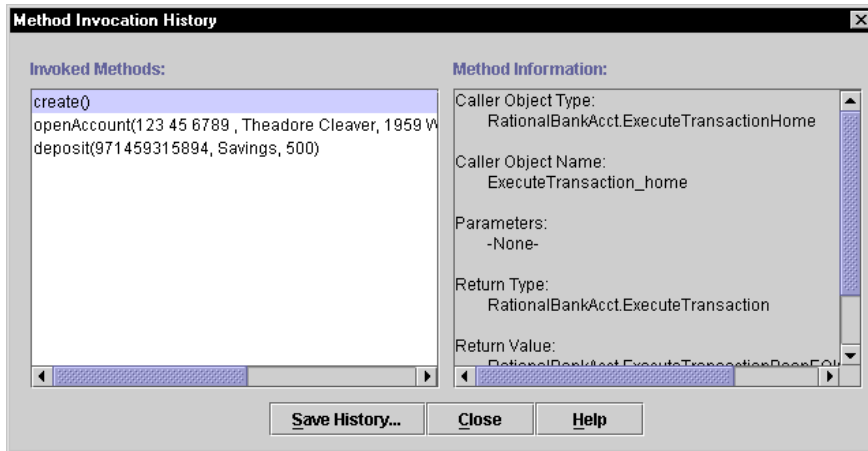
7 Click **Create**.

The deposit amount appears in the **Parameter Value** box. The named object appears in the **Name** box.

8 Click **Invoke** to deposit the specified amount into the account.

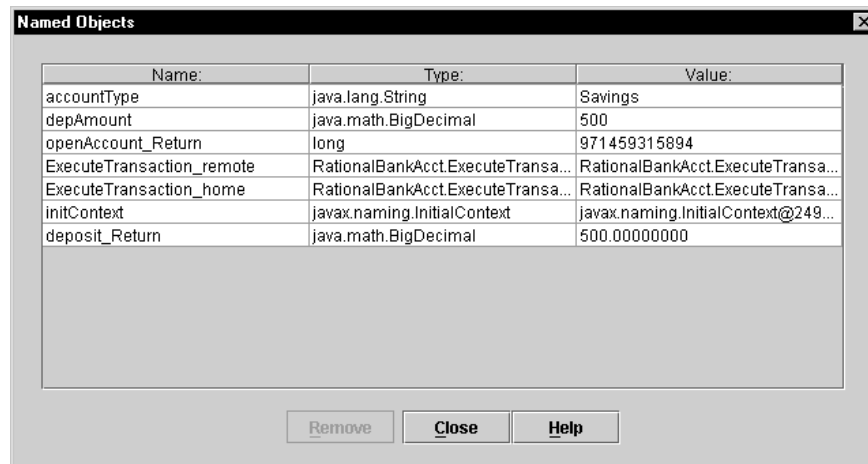
9 Click **View > Method History** to see a history of the invoked methods.

The history includes methods that are successful, as well as those that have failed to execute.



When you finish viewing the Method History, click **Close**.

10 Click **View > Objects** to see a list of the named objects that were created. The named objects are the ones that can be reused during the session.



When you finish viewing the objects, click **Close**.

Invoking the `getBalance()` method

To invoke the `getBalance()` method:

- 1 Click the **`getBalance()`** method from the **Methods** list.
- 2 Right-click on the first Value box in the Parameters pane and select `openAccount_Return`.

The `openAccount_Return` object was created automatically when the `openAccount` method was invoked.

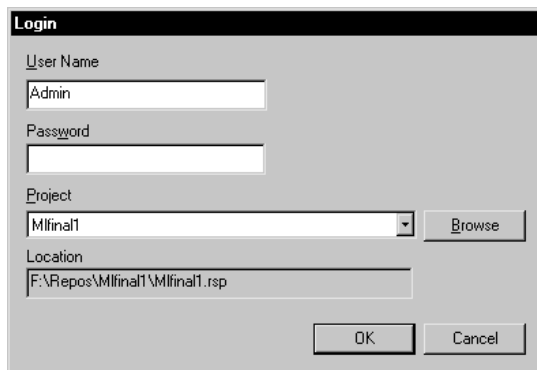
- 3 Right-click on the second Value box in the Parameters pane and select **`accountType`**.
- 4 Click **Invoke**.

The current account balance should now appear in the **Last Return Value** box.

Inserting a Verification Point

To insert a verification point:

- 1 Click **Insert > Verification Point**.
- 2 If prompted, log in to a project and click **OK**.

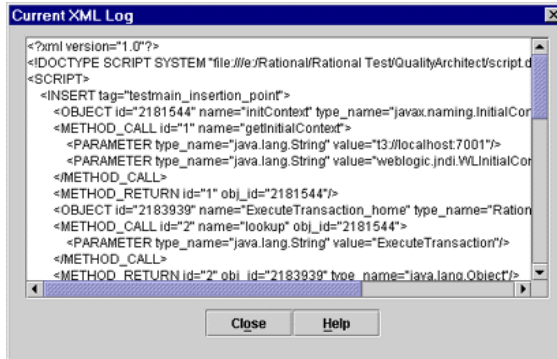
A screenshot of a 'Login' dialog box. It has a title bar 'Login' and a grey background. There are four input fields: 'User Name' with 'Admin' entered, 'Password' (empty), 'Project' with a dropdown menu showing 'Mlfinal1' and a 'Browse' button to its right, and 'Location' with 'F:\Repos\Mlfinal1\Mlfinal1.rsp' entered. At the bottom are 'OK' and 'Cancel' buttons.

- 3 When the Select Verification Point Type dialog box appears, select **DatabaseVP**, or another type of verification point that you have implemented.
- 4 Type a name for the Verification Point and click **OK**.
- 5 At this point, the Query Builder wizard starts. Use the Query Builder to connect to the database and to define a Select statement that can be used to query the database. For more information about the Query Builder, see *Using the Java Query Builder to Add Verification Points* on page 59.

Viewing the XML Log

To view the XML log of the current session:

- 1 Click **View > XML Log**.



- 2 When you finish viewing the log, click **Close**.

Generating a Test Script from the XML Log

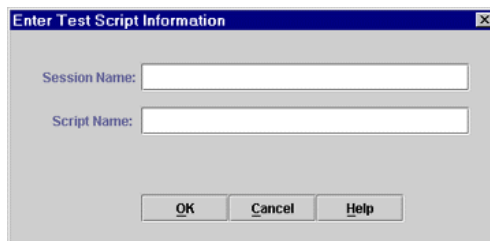
To stop the session and generate a test script:

- 1 Click **Stop**.

You are asked whether or not you want to generate a test script from the XML log.

- 2 Click **Yes** to generate the test script.

Note: If you are not already logged into a project, you are now prompted to log in. Then, you are prompted to enter a Session Name and a Script Name.



- 3 Enter a name for the current session in the **Session Name** box.
- 4 Enter a name for the generated script in the **Script Name Box** and click **OK**.

The script is saved in Test DataStore under `recordedtests\<session name>\<scriptname.java>`.

This chapter provides the information you need to use Rational QualityArchitect to test COM-based component applications. Topics include:

- Overview
- Requirements for testing COM components
- Working with the sample model
- Executing test scripts
- Using COM scenario tests to test transactions
- Using the OLE DB Query Builder to add verification points

Overview

With QualityArchitect, you can test COM components after they have been built, or you can test iteratively during the development process.

Testing Existing Objects

To test completed COM components, you can use the type library import tool in Rose. The type library import tool defines 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 Rose ATL (ActiveX Type Library) object creation wizard.

By default, the import tool does not capture the methods on each interface. You can capture the methods afterwards in one of two ways:

- By doing a full import from the Rose Component view
- By doing a full import of selected interfaces from the Logical view

Note: The methods must be imported into the model in order to model method messages in sequence diagrams.

For more information about testing completed COM components, see the topic *Importing Type Libraries Into the Model* in the Rose Help.

Testing with Iterative Development

To test iteratively, you can use the Update Code feature of Rose to create new classes for your component and then round-trip the code that you add to these classes into the model as you develop.

When you are ready to test, you import the type library for your component and then model the transactions in interaction diagrams using the interfaces imported from the type library.

When you change your code, either by updating it from the model or directly, you simply refresh the type library information in the model by re-importing it.

Programming in Visual Basic

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.

QualityArchitect treats each test script generated for COM as a Visual Basic project. Each project consists of several files, as follows:

- *test.vbp* – The project file. This is what you open in Visual Basic.
- *test.bas* – The main program that calls the test script program in the .cls file. Visual Basic requires a main program to begin execution.
- *test.cls* – The actual test script.
- *test.re* – A standard Visual Basic resource file used to store datapool configuration information. This file can be edited with the Visual Basic 6 resource file editor add-in. This file needs to be edited only if you plan to execute test scripts through Rational TestManager.

Every time you generate a unit test or a scenario test, QualityArchitect uses templates to produce these files.

Requirements for Testing COM Components

To test COM components with QualityArchitect, you need the following:

- Visual Basic 6.0
- A model that represents your COM interfaces and coclasses

You can obtain this model by importing a type library.

In addition, QualityArchitect must be able to find the Visual Basic resource compiler file, `rc.exe`. QualityArchitect searches for it on the System or User path and then looks in one of the following typical installation locations:

- `<Visual Studio Directory>\Common\MSDev98\Bin`
- `<Visual Studio Directory>\VB98\Wizards`

If QualityArchitect fails to find `rc.exe`, you are prompted to supply the path.

You can also specify the path to `rc.exe` in the Rational QualityArchitect Options dialog box. To do so:

- 1 In Rose, click **Tools > QualityArchitect > Console**.
- 2 Click the **Edit Rational QualityArchitect Options** button.
- 3 Click the **VB Options** tab.
- 4 Specify the location of `rc.exe` in the Resource Compiler text box.

Working with the Sample Model

The installation procedure installs a sample Rose model—`rqacomsample`—that you can use to try out QualityArchitect. This model reflects a traditional transaction processing system that allows users to credit or debit their accounts on a server.

The `rqacomsample` model already contains an imported type library and a component associated with a Visual Basic source project.

Understanding the Component View

Rose's component view shows the physical pieces of software that are included in the model, for example:

- A COM object (RQACOMSample Ver 1.0) that is associated with the imported type library
- An optional ActiveX DLL (RQACOMSample) that is associated with the Visual Basic source project

This DLL is used for round-trip engineering.

Other components, such as `Stdole` and `DAO`, are included with COM.

Understanding the Logical View

The Logical View shows the packages, classes, interfaces, and operations in the model, such as:

- RQACOMSample (from COM)

The package that is created when the type library is imported. (In the sample model, the type library has already been imported.) This package contains several COM interfaces—_Account, _UpdateReceipt, _GetReceipt, and _MoveMoney.

- RQACOMSample (from Reverse Engineered)

The package containing objects used in round-trip engineering of the source code.

For script generation, QualityArchitect uses the COM package.

Generating Test Assets

This section shows you how to generate the various test assets you need to test COM components with QualityArchitect. Test assets include test scripts, datapools, stubs, and lookup tables.

Note: In a model it is possible to have more than one CoClass that realizes the same interface. In this case, when you generate a unit test script, QualityArchitect displays a dialog box that prompts you to specify the CoClass that you want to use.

Generating Unit Test Scripts from a Rose Model

You can use the `rqacomsample` sample model that is installed with QualityArchitect to generate unit test scripts from Rose. For example, try generating a unit test for the `Perform` method, which is part of the `_MoveMoney` interface.

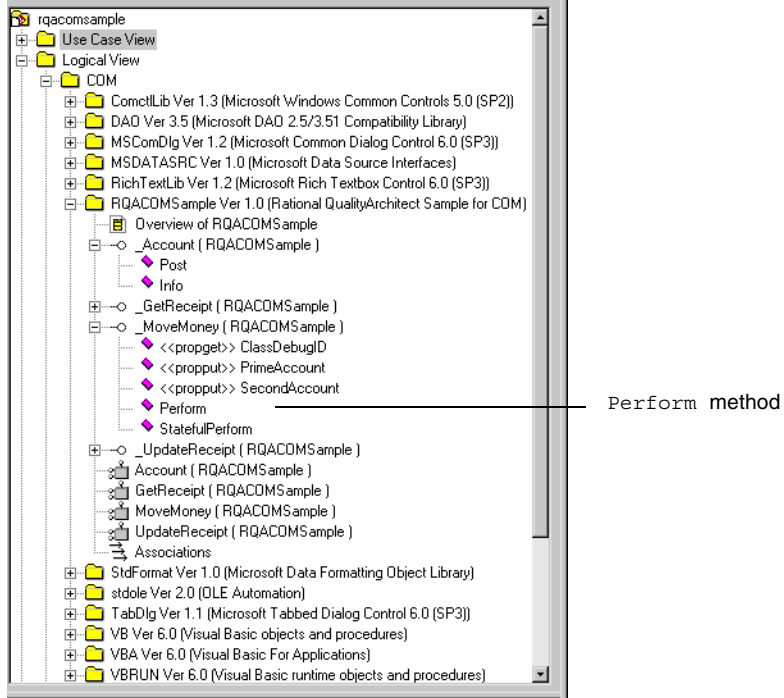
To generate the unit test:

- 1 If you haven't already done so, create a project with the Rational Administrator for maintaining your test assets.

For details, see *Adding a Project* in the Administrator Help.

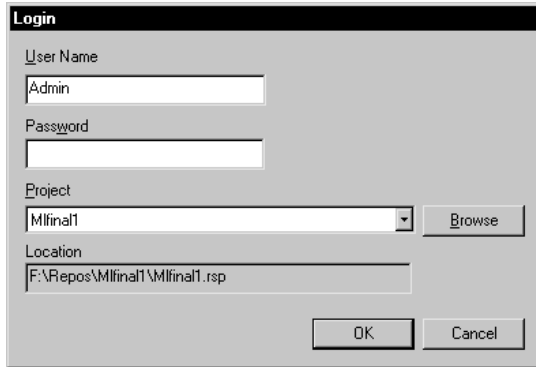
- 2 Start Rose and open the `rqacomsample.mdl` model.

- 3 Optionally, right-click the `Perform` method and click **QualityArchitect > Select Unit Test Template**.



- 4 Optionally, select the **TestName.vbp** template and click **Open**.
For more information about templates, see *Templates* on page 7.
- 5 In Rose, right-click the `Perform` method again and click **QualityArchitect > Generate Unit Test**.

A message appears, indicating that code generation is in progress, after which you will be prompted to log in to a Rational project.



- 6 Log in to the project and click **OK**.

Each project contains a datastore for storing test assets, such as datapools, lookup tables, and log files.

- 7 If this is the first time generating a script for a particular datastore, you are prompted to select a directory in which to store your scripts.

Select a directory location and click **OK**.

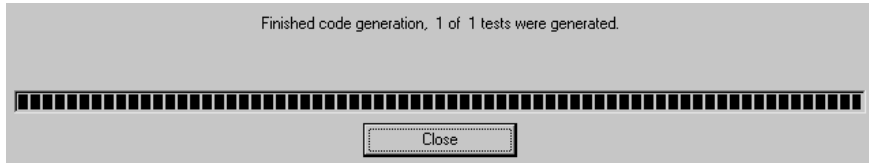
QualityArchitect creates a directory hierarchy under the location you have chosen and saves several files, as follows:

- MoveMoneyPerform.vbp
- MoveMoneyPerformMain.bas
- Perform.cls
- MoveMoneyPerform.res

The Visual Basic project file that is created (MoveMoneyPerform.vbp) is assigned a name of the format *InterfacenameMethodname*.

Note: In situations where there is more than one CoClass realizing an interface, you are prompted to select a CoClass.

- When code generation is complete, click **Close** to close the progress bar.



Note: QualityArchitect maintains an association between the test script directory that you have chosen and the test assets that are stored in the project's test datastore.

Creating a Datapool

A *datapool* is a set of records that you can use to drive a test script. Datapool support is included automatically in each test script generated from Rose.

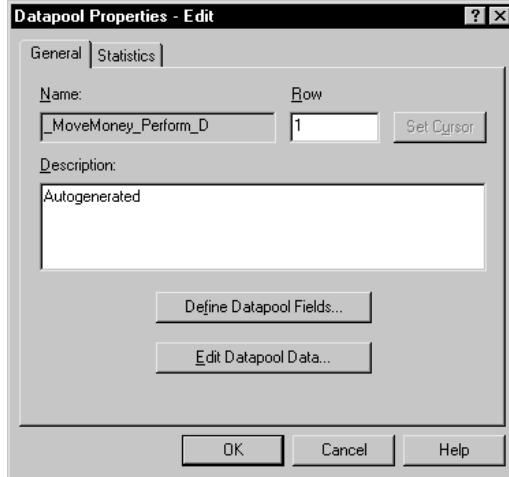
The following code fragment shows the datapool name and the parameter (column) names embedded in the test script:

```
'Open Datapool
  dp.Open "_MoveMoney_Perform_D" _____ Datapool name
'Loop over datapool and perform test.
  While dp.Fetch
    'Keep counter of number of rows fetched.
    NumRows = NumRows + 1
    'Get the column data from the datapool.
    lPrimeAccount = dp.Value("lPrimeAccount")
    lSecondAccount = dp.Value("lSecondAccount")
    lAmount = dp.Value("lAmount")
    lTranType = dp.Value("lTranType")
    expRet = dp.Value("expectedReturn")
    expErr = dp.Value("expectedError")
```

Column name

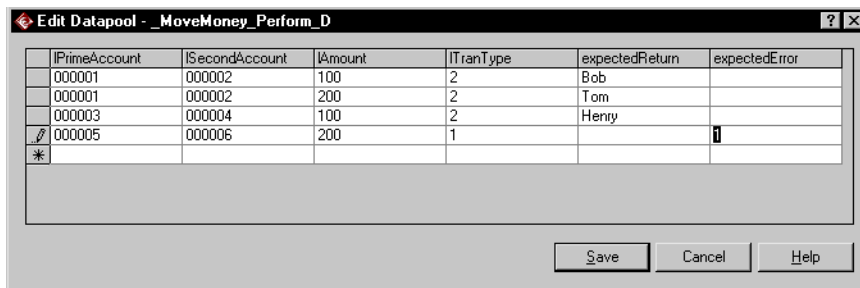
To create a datapool that you can use for testing the Perform method:

- 1 In the Rose browser, right-click the Perform method and click **QualityArchitect > Create Datapool**.



QualityArchitect auto-generates a datapool, using the parameters in the Perform method (IPrimeAccount, ISecondAccount, IAmount, ITranType, expectedReturn, expectedError) for the column names. It names the datapool `_MoveMoney_Perform_D`.

- 2 Click **Edit Datapool Data** and populate the datapool with several rows.



For more information about datapools, see *Datapools* on page 21, the *Test Script Services for Visual Basic* manual, and the online Help for Rational TestManager.

Generating Stubs for the Unit Test

With QualityArchitect, you can create stubs for any component called by the method-under-test.

When you generate stubs, QualityArchitect creates Visual Basic class files that you can copy into your Visual Basic project and use in place of actual components. Simply replace the actual component with the stub and recompile the project.

Because the `MoveMoney` method calls methods in the `Account` interface class, you can either generate a stub or run the tests directly against the actual `Account` class.

To generate the stubs:

- 1 In the Rose browser, right-click the **Account** class and click **QualityArchitect > Generate Stub**.
- 2 Select a directory for storing the stubs.
- 3 Click **OK**.
- 4 Make a copy of the `Account` class.
- 5 Replace the real `Account` class in the `RQACOMSample` application with the stubbed version.
- 6 Add a reference to “Rational QualityArchitect Playback Type Library” in the `RQACOMSample` project in Visual Basic.
- 7 Comment out any lines in the code for complex objects.
- 8 Recompile `RQACOMSample`.

Note: Stubs must be deployed on the same computer as the test script.

For a high-level overview of stubs, see *Stubs* on page 16.

Creating Lookup Tables

After you generate the stubs, you need to create a lookup table for each method in the stub called by the method-under-test. In this case, you need to create a look up table for the `Post` method.

Lookup tables are based on Rational datapool technology. Whereas a datapool is used to test inputs and expected behavior, a lookup table is used with stubs to simulate the behavior of an actual component.

To create a lookup table for the `Post` method in the `Account` class:

- 1 From the Rose browser, select the `Post` method in the `Account` bean.
- 2 Click **Tools > QualityArchitect > Stubs > Create look-up table**.
- 3 Populate the table with real data, for example:

IAccountNo	IAmount	Expected Return	Expected Error
<i>Generated ID 1</i>	<i>amount of Post</i>		
<i>incorrect AccountNo</i>			1003

At this point, you should be finished creating your tests assets.

Executing Test Scripts

To execute your text script in Visual Basic:

- 1 Open the Visual Basic project file.
- 2 Edit the test script (the .cls file) as needed.
- 3 Click **Run > Start with Full Compile**.

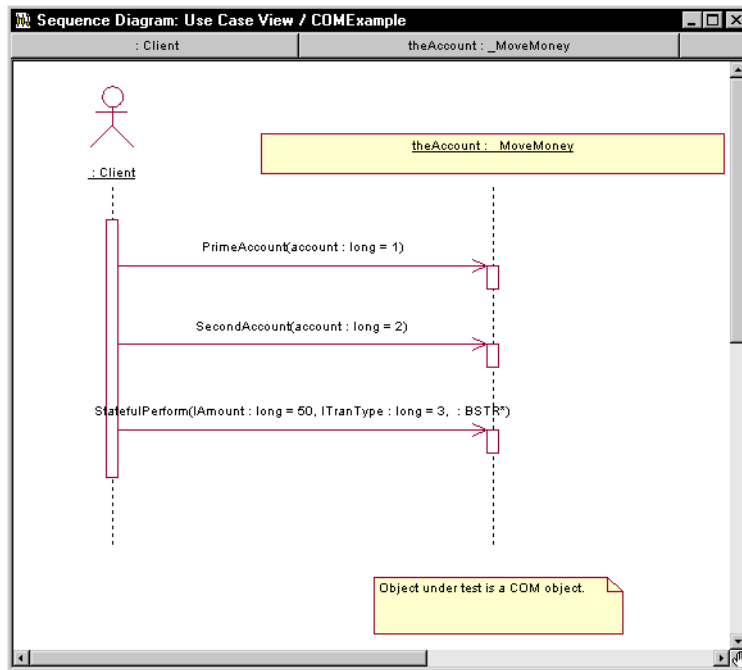
You can also execute test scripts from TestManager. Before doing so, be sure that Visual Basic 6.x is included on your system path.

To execute your test script in TestManager:

- 1 Start TestManager.
- 2 Click **File > Run Test Script > RQA VB Scripts**.
- 3 Select the script and click **Open**.
- 4 From the Run Script dialog box, click **OK**.

Using COM Scenario Tests to Test Transactions

Scenario tests use Rose interaction diagrams to test transactions. To try out this feature, you can generate a scenario test for the COMExample sequence diagram (see the following figure) that is included in the rqaconsample model.

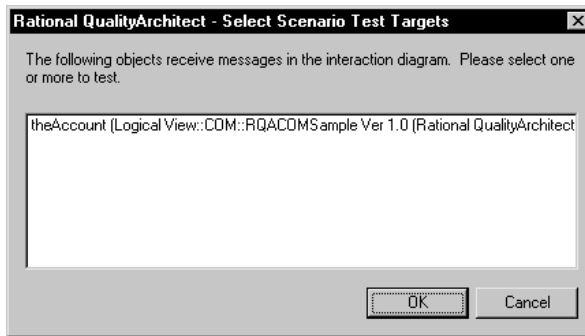


When you generate a scenario test, QualityArchitect prompts you to insert a verification point for each message in your interaction diagram.

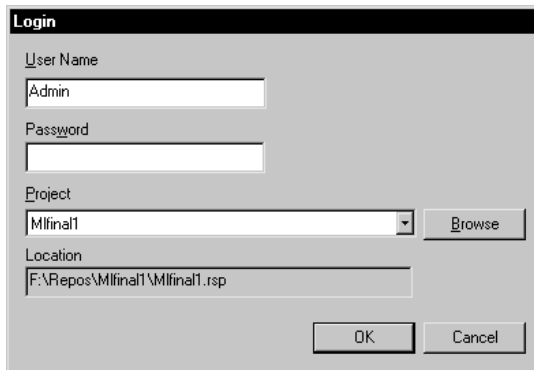
To generate a scenario test:

- 1 Open the COMExample sequence diagram in Rose.
- 2 Right-click in the diagram and click **QualityArchitect > Select Scenario Test Template**.
- 3 Verify that `com_scenario_script_template` appears in the File Name box, and then click **Open**.
This template resides in the `QualityArchitect\Templates\Scenario Test Templates\COM VB` directory.
- 4 Right-click in the interaction diagram and click **QualityArchitect > Generate Scenario Test**.
- 5 In the Select Scenario Test Targets dialog box, select the scenario test targets—that is, the objects you want to test—and click **OK**.

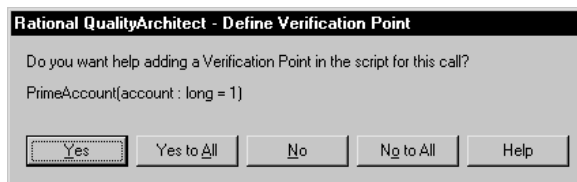
You can select one or more objects to test.



- 6 If prompted, log in to a project and click **OK**.

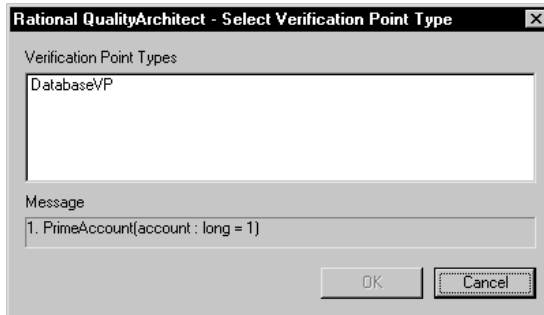


- 7 In the Define Verification Points dialog box, click **Yes** to add a verification point for the first message in the diagram—getBalance.



- 8 In the Select Verification Point Type dialog box, select a verification point type and click **OK**.

For this release, select the database verification point type—DatabaseVP.

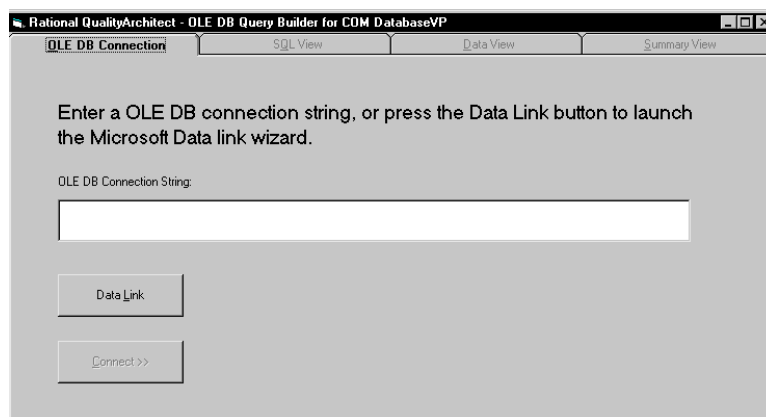


- 9 At this point, the OLE DB Query Builder wizard starts. Use the Query Builder to connect to the database and to define a Select statement that can be used as a query. For more information about the Query Builder, see *Using the OLE DB Query Builder to Add Verification Points* on page 99.
- 10 Repeat Steps 8 and 9 for each message in the diagram.
- 11 After the last message, you are prompted to add a verification point at the end of the scenario. Click **Yes**.
- 12 Select a directory to store the tests in and click **OK**.

Note: You can also start the scenario test generator by right-clicking on a diagram in the Rose browser and then clicking **QualityArchitect > Generate Scenario Test**.

Using the OLE DB Query Builder to Add Verification Points

The OLE DB Query Builder is a tool that helps you connect to and interact with databases for the purpose of defining database verification points.



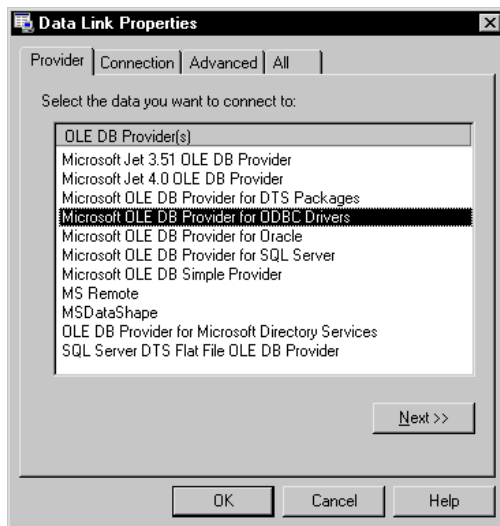
Connecting to the Database via OLE DB

The first step involved in building a custom SQL query is to enter an OLE DB connection string, which allows you to connect to a database.

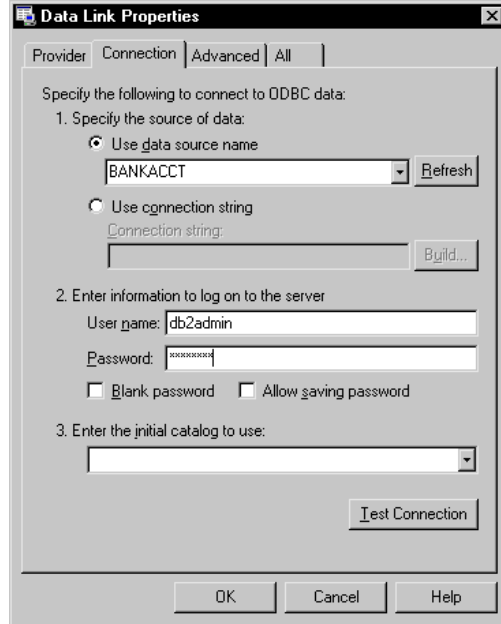
When the OLE DB Query Builder starts, you can either type the connection string manually or click the **Data Link** button to display the Data Link Properties wizard, a graphical user interface that will assist you in building the connection string.

To use the Data Link Properties wizard:

- 1 Click the **Data Link** button to display the Provider page of the Data Link Properties wizard. (Click **Help** at any point to view online Help for the wizard.)
- 2 On the Provider page, select the appropriate OLE DB provider and click **Next**.



- 3 On the Connection page, select a data source name and enter a user name and password, if these are required to log in to the server.



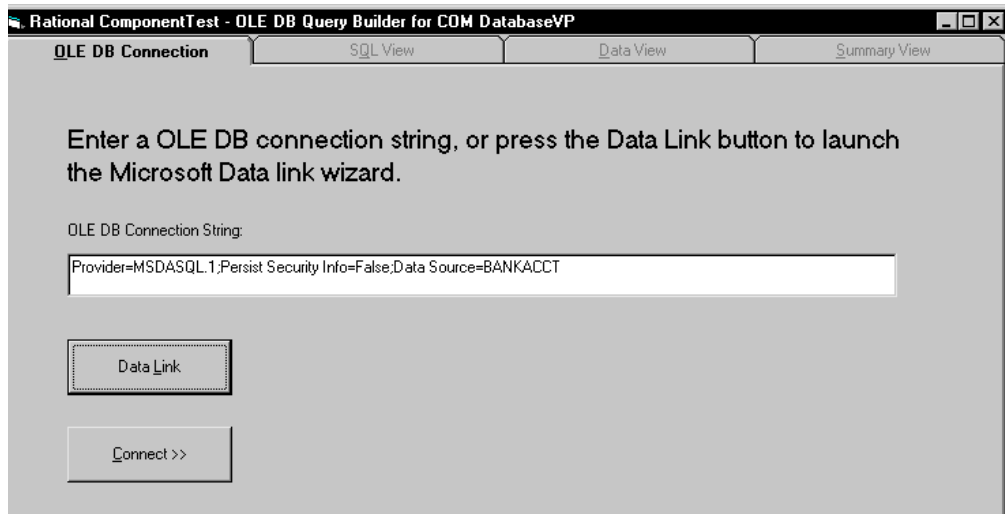
- 4 Also, on the Connection page, check **Allow Saving Password** to allow the password to be included in the connection string.

Note: If saved, the password is not encrypted.

- 5 Click **Test Connection**.

In most cases, it should not be necessary to change any of the default settings on the **Advanced** or **All** pages. Click the **Help** button for details.

6 Click **OK** to save the OLE DB Connection String and redisplay the Query Builder.



7 Click **Connect** to connect to the database.

Designing a Custom SQL Statement

After you connect to the database, you can design the custom SQL query statement that will be used to retrieve specific data from the database.

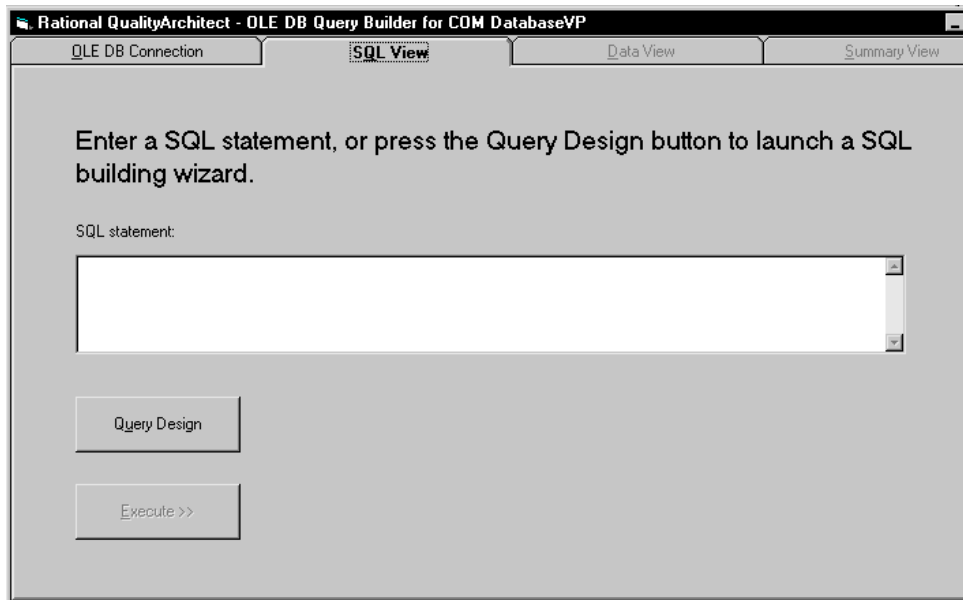
If you are familiar with SQL syntax and are familiar with the schema of the database you are connecting to, you can simply enter your custom SQL query statement in the SQL text box.

Using the Query Design Wizard

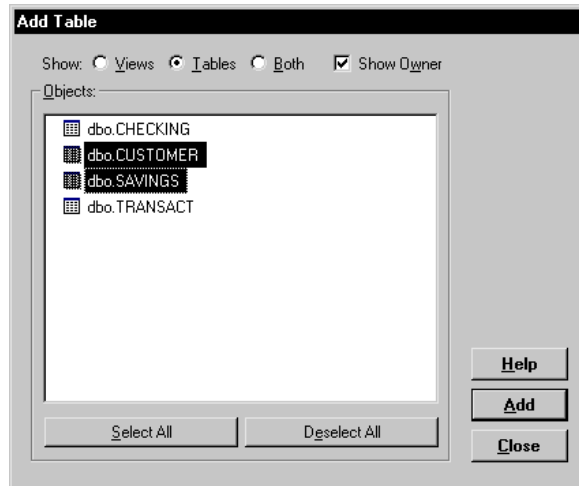
Alternatively, you can use the Query Design wizard to interactively walk you through the design of your custom SQL query statement. The Query Design wizard helps you easily create complex SQL query statements by taking you step-by-step through the design process.

To use the Query Design wizard:

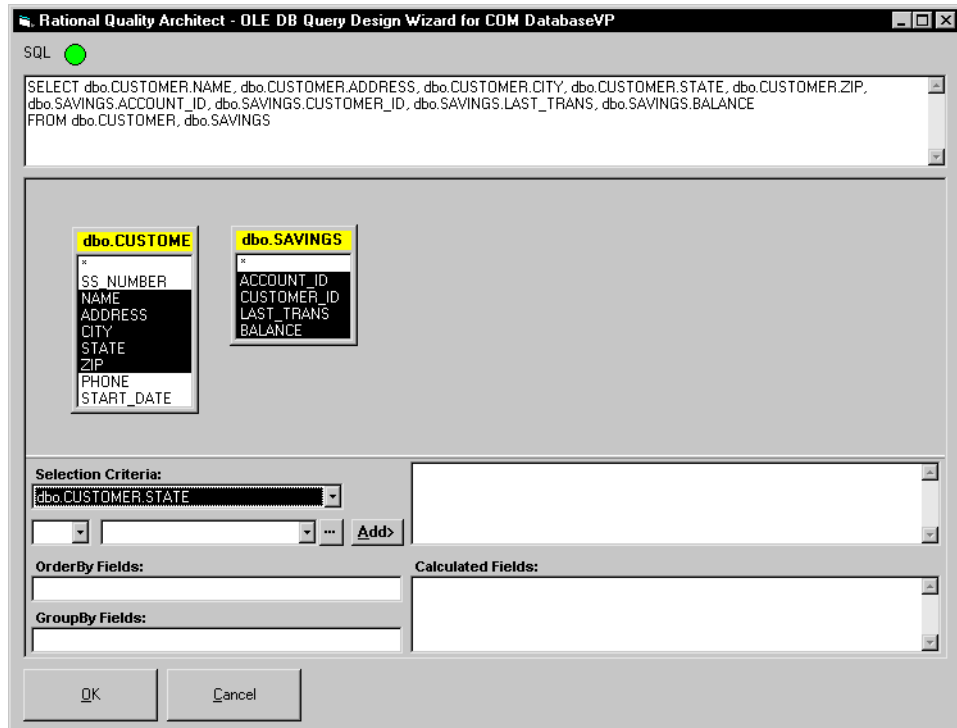
- 1 Click the **Query Design** button.



- 2 Select the tables you want to query and click **Add**.



- 3 Select the columns to include in the result set.



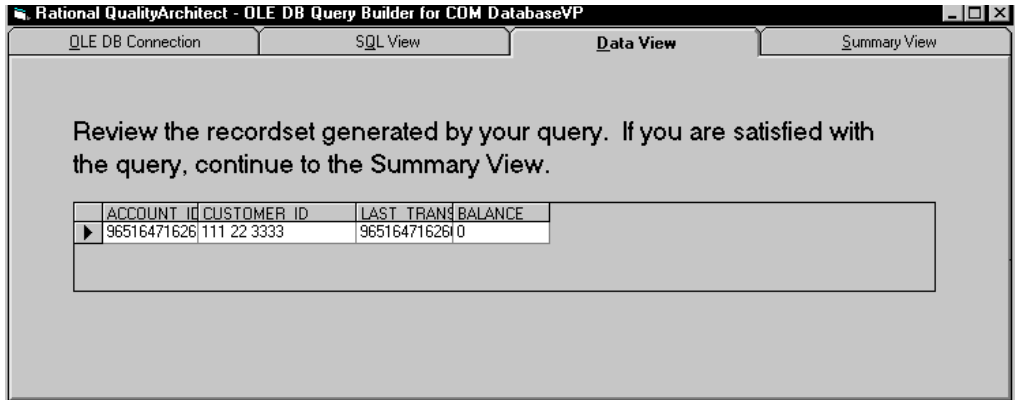
Column names are listed in *TableName.ColumnName* format so that you can easily identify the columns in each table.

- 4 Enter selection criteria that will restrict the returned query results (optional).
- 5 Enter a sort order in the **OrderBy Fields** box (optional).
- 6 Enter a GroupBy order in the **GroupBy Fields** box (optional).
- 7 Enter any calculated fields.
- 8 Click **OK**.
- 9 Verify the SQL statement, make any edits as needed, and click **Execute**.

Note: The round SQL button turns green if the SQL statement is syntactically correct. Otherwise, the button turns red.

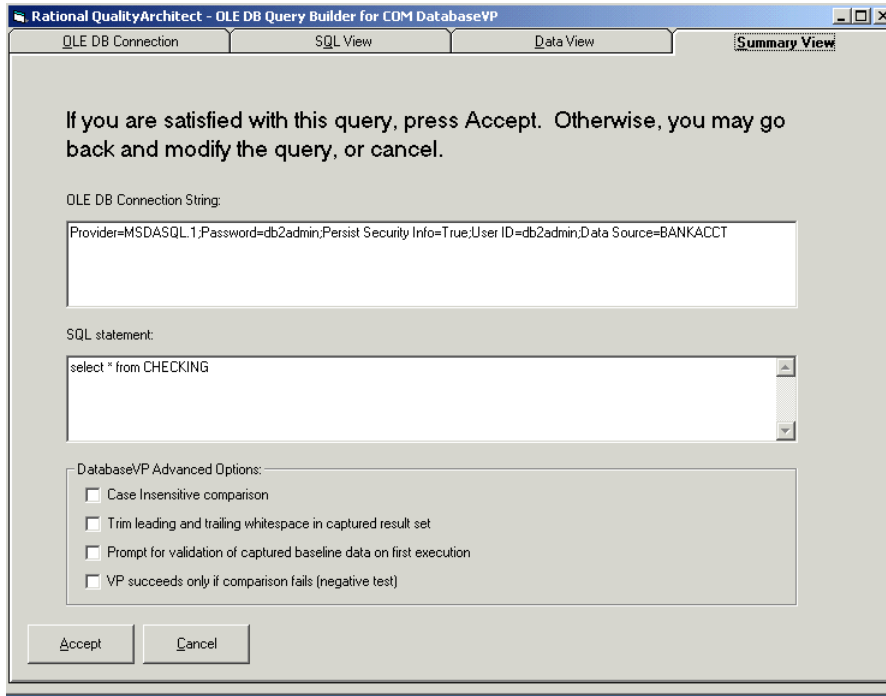
Reviewing the Record Set

The next step in building the query is to review the record set returned by the SQL statement. If you are satisfied with the results, click the **Summary View** tab. Otherwise, click the **SQL View** tab and redesign the query.



Accepting the Query

The Summary View shows the OLE DB Connection String and the SQL statement that you have created. Click **Accept** to generate the query and complete the verification point.



DatabaseVP Advanced Options

The Summary View also includes a number of advanced options that you can set for the database verification point. These options include:

Option	Description
Case Insensitive comparison	Specifies that the verification should be case insensitive. By default, text comparisons are case sensitive.
Trim leading and trailing whitespace in captured result set	Specifies that captured values should have whitespace trimmed from the right and left sides
Prompt for validation of captured baseline data on first execution	Specifies that the first run of a static verification point should display the captured data for the tester to validate before storing it as the expected (baseline) data object.
VP succeeds only if comparison fails (negative test)	Specifies that the verification point's expected result is failure. If the comparison fails and this option is set, the verification point succeeds.

Template Replacement Variables

A

This section describes the replacement variables that are used in the various templates used by Rational QualityArchitect. The code generators supplied with QualityArchitect replace the variables in the templates with real code and data derived from Rose model elements.

These replacement variables can be logically grouped into the following general categories:

- Replacement variables for unit test generation templates
- Replacement variables for scenario test generation templates
- Replacement variables for stub generation templates

Replacement Variables for Unit Test Generation Templates

The replacement variables for the unit test generation templates can be logically divided into the following categories:

- Variables used in all languages
- Variables used only in Visual Basic and COM
- Variables used only with Enterprise JavaBeans (EJBs)

For a list of the unit test generation templates, see *Templates* on page 7.

Variables Used in All Unit Test Generation Templates

The following table lists the replacement variables that are used in all of the unit test generation templates:

Variable	Description and Example
<generation_date>	Shows when the script was generated. <generation_date> in the template becomes Date: 9/19/00 2:56:32 PM in the script.
<root_package>	Root directory location for all generated unit test scripts. The <root-package> is always <code>unittests</code> . <root_package> in the template becomes <code>unittests</code> in the script.
<package_name>	The package hierarchy in the Rose model, excluding the class and the top-level package name. For example, the Rose Item Fully Qualified Name := Logical View::COM::MyComponent::MyClass::MyOperationUnderTest" TestName := MyClassMyOperationUnderTest tms.StartTestServices "<root_package>\<package_name>\<test_name>" becomes <code>unittests\COM\MyComponent\MyClassMyOperationUnderTest</code> " in the script.
<operation_name>	The name of the operation selected for generation. Class=<operation_name>; TestNameScript.cls becomes Class=MyOperationUnderTest; TestNameScript in the script.
<class_name>	The name of the class selected for generation. Template: <class_name> Script: MyInterface
<test_name>	The name of the test script, calculated as follows: ClassNameOperationName QualityArchitect may modify <test_name> to conform to the target generation language. Example: ClassName := "_Account", OperationName := "Post", Language := "VB" Before: ExeName32="<test_name>.exe" After: ExeName32="AccountPost.exe" Note: In VB an object name cannot begin with "_" character, so the "_" is removed at generation time.

Variable	Description and Example
<datapool_name>	<p>The name of the datapool, calculated as follows: ClassName_OperationName_CTD_DATAPOOL</p> <p>The DataStore understands that datapool names are limited to 40 characters, so it takes two parameters and tries to figure out the best name, such as "ClassNameOperationName", that fits within the 40 character limit. CTD_DATAPOOL causes "_D" to be appended to the end of the name and CTD_LOOKUPTABLE causes "_L" to be appended to the end of the name.</p> <p>Example: ClassName := "_Account", OperationName := "Post", Language := "VB" Before: dp.Open("<datapool_name>") After: dp.Open("_Account_Post_D")</p>
<index>	<p>Unique index, calculated as follows: ScriptSource\PackagePath\TestName</p> <p>CTD_SCRIPTTYPE_VB - Uses the "RQA VB Test Scripts" script source for the index calculation. CTD_SCRIPTTYPE_JAVA - Uses the "RQA Java Test Scripts" script source for the index calculation.</p> <p>Example: TestName := "AccountPost", PackagePath := "unittests\COM\MyComponent", ScriptType := CTD_SCRIPTTYPE_VB</p> <p>If no files exist with the name <ScriptSourceDirectory>\<PackagePath>\TestName Before: ExeName32="<test_name><index>.exe" After: ExeName32="AccountPost.exe"</p> <p>If <ScriptSourceDirectory>\<PackagePath>\AccountPost.vbp exists Before: ExeName32="<test_name><index>.exe" After: ExeName32="AccountPost1.exe"</p>
<check_expected_result>	<p>The code generated here depends on the target language for generation and the return value of the operation. If the operation return type is void, or if VB is marked as a "Sub", then the actual code that is inserted is as follows:</p> <p>If (operation does NOT have a return value) Then <check_expected_result> := 'Log message indicating success. Else <check_expected_result> := 'If statement to compare the expected return with the actual return for equivalence. 'If equal, then log success, otherwise log error.</p>

Variable	Description and Example
<method_declaration>	The code that is generated here depends on the target language for generation and the operation that is selected. The result is a string that can be used to declare a method. No newline characters are added to this variable.
<parameter_declarations>	The code that is generated here depends on the target language for generation and the operation that is selected. The result is a string that can be used to declare all of the variables that will be used as arguments to the operation that has been selected for generation. Newline characters are added to this string so that each variable is declared on a new line.
<parameter_initialization>	The code that is generated here depends on the target language for generation and the operation that is selected. The result is a string that can be used to initialize all of the variables that are declared in <parameter_declarations> to be values from a datapool. Each parameter type is checked to see if it can be driven using a datapool value. If not, a comment is inserted in place of the datapool assignment.
<return_val>	The code that is generated here depends on the target language for generation and the operation that is selected. The result is an empty string if the operation does NOT return a value, or "actRet = " if it does.
<operation_arglist>	<p>The code that is generated here depends on the target language for generation and the operation that is selected. The result is a string that can be used in the function invocations string.</p> <p>Example</p> <pre>TestClass::TestShort(ByVal Arg1 As Integer, ByVal Arg2 As Integer) As Integer Before: Dim tc As TestClass Dim actRet As Integer <parameter_declarations> ' Invoke: <method_declaration> of TestClass <return_value> tc.<operation_name> (<operation_arglist>) After: Dim tc As TestClass Dim actRet As Integer Dim arg1 As Integer Dim arg2 As Integer ' Invoke: TestShort(ByVal Arg1 As Integer, ByVal Arg2 As Integer) As Integer of TestClass actRet = tc.TestShort (arg1,arg2)</pre>

Variables Used Only with the COM/Visual Basic Templates

The following table lists the replacement variables that are used only with the COM/VB unit test generation templates:

Variable	Description and Example
<interface_name>	<p>Interface name of the operation selected for generation.</p> <p>Example: Rose Item Fully Qualified Name := "Logical View::COM::MyComponent::_MyInterface::MyOperationUnderTest"</p> <p>Before: <Interface_name> After: _MyInteface</p> <p>Note: Same as <class_name> for VB scripts generated from COM interfaces.</p>
<library_name>	<p>Library name that contains the class that is implementing this operation, calculated as:</p> <pre>Dim rsModule As RoseModule rsModule = cls.GetAssigendModules(1) Dim rsProp As RoseProperty rsProp = rsModule.FindProperty("COM","library") <library_name> = rsProp.Value</pre> <p>Example: Rose Item Fully Qualified Name := "Component View::COM::MyComponent"</p> <p>Before: Set obj = CreateObject("<library_name>.Foo") After: Set obj = CreateObject("MyComponent.Foo")</p>
<coclass_name>	<p>CoClass that implements the interface that the operation belongs to. If more than one coclass implements the interface, the user is prompted to choose a CoClass.</p> <p>Example: Rose Operation := "Logical View::COM::MyComponent::_Account::Post" Rose Interface := "Logical View::COM::MyComponent::_Account" Rose CoClass := "Logical View::COM::MyComponent::Account"</p> <p>Before: Set obj = CreateObject("MyComponent.<coclass_name>") After: Set obj = CreateObject("MyComponent.Account")</p>

Variable	Description and Example
<coclass_guid>	<p>CoClass GUID that implements the interface that the operation belongs to. If more than one coclass implements the interface, the user is allowed to select the coclass to use.</p> <p>Example: Rose Operation := "Logical View::COM::MyComponent::_Account::Post" Rose Interface := "Logical View::COM::MyComponent::_Account" Rose CoClass := "Logical View::COM::MyComponent::Account"</p> <p>Before: Set obj = createObject(<coclass_guid>) After: Set obj = createObject(87DD307A-01DD-42BD-997D-967E5CBB3281)</p>
<coclass_progid>	<p>Coclass_progid, or programmatic identifier, is a registry entry that is associated with a CLSID. Like the CLSID, the coclass_progid identifies a class, but with less precision.</p> <p>Example: Rose Operation := "Logical View::COM::MyComponent::_Account::Post" Rose Interface := "Logical View::COM::MyComponent::_Account" Rose CoClass := "Logical View::COM::MyComponent::Account"</p> <p>Before: Set obj = CreateObject("<prog_id>") After: Set obj = CreateObject("MyComponent.Account")</p>
<project_reference>	<p>Project reference allows you to add an object or type library or project references to your project. This makes another application's objects available in your code. Once a reference is set, the referenced objects are displayed in the Object Browser.</p> <p>Before: <project_reference> After: Reference=*\\G{F6E99F53-33E7-4478-A5F7-E2801010E673}#1.0#0#RQACOMSample.dll#Rational QualityArchitect Sample for COM</p>

Variables Used Only with the EJB Templates

The following table lists the replacement variables that are used only with the EJB unit test generation templates:

Variable	Description and Example
<remote_interface_name>	<p>Name of the EJB Remote Interface</p> <p>Rose Item Fully Qualified Name := Logical View::RationalBankAcct::Checking::getBalance</p> <p>Before: Object o = initContext.lookup("<remote_interface_name>"); After: Object o = initContext.lookup("Checking");</p>

Variable	Description and Example
<home_interface_name>	<p>Name of the EJB Home Interface Rose Item Fully Qualified Name := Logical View::RationalBankAcct::CheckingHome::create</p> <p>Before: <home_interface_name> home = (<home_interface_name>) PortableRemoteObject.narrow(o, <home_interface_name>.class); After: CheckingHome home = (CheckingHome) PortableRemoteObject.narrow(o, CheckingHome.class);</p>
<primary_key_class_name>	<p>Name of the EJB Primary Key class Rose Item Fully Qualified Name := Logical View::RationalBankAcct::CheckingKey</p> <p>Before: <primary_key_class_name> key = new <primary_key_class_name>(<key_params>); After: CheckingKey key = new CheckingKey(<key_params>)</p>
<createparam_declaration>	<p>String that declares all of the parameters for the EJB Home Interface Create method that is associated with the operation's class selected for generation.</p> <p>Example: Before: // Declare arguments to the create method <createparam_declaration> After: // Declare arguments to the create method long accountID = 0; String customerID = null; long lastTrans = 0; java.math.BigDecimal openBalance = null;</p>
<createparam_init>	<p>String that will initialize all of the create parameters for the EJB Home Interface Create method using a datapool initialization.</p> <p>Before: // Declare arguments to the create method <createparam_declaration></p> <p>After: Initialize arguments for the create method AccountID = dp.value("accountID").longValue(); customerID = dp.value("customerID").toString(); lastTrans = dp.value("lastTrans").longValue(); openBalance = dp.value("openBalance").getBigDecimal();</p>

Variable	Description and Example
<create_params>	String that can be used as an arglist for the create method invocation. Before: Invoke the create method. <remote_interface_name> remote = home.create(<create_params>); After: Invoke the create method. Checking remote = home.create(accountID, customerID, lastTrans, openBalance);
<keyvalue_declaration>	Same as <createparam_declaration> but for the Key class constructor method that is associated with the operation class selected for generation.
<keyvalue_init>	Same as for the <createparam_init> but for the key class constructor's parameters.
<key_params>	Same as for the <create_params> but for the key class constructor's parameters.
<datapool_init>	Same as the <parameter_initialization> that is indicated above for COM/VB. This is provided only for EJB and NOT for COM, whereas <parameter_initialization> is provided for both.

Replacement Variables for Scenario Test Generation Templates

The replacement variables for the scenario test generation templates can be logically divided into the following categories:

- Variables used in both the COM/VB and EJB templates
- Variables used only in the COM/VB templates
- Variables used only in the EJB templates

For a list of the scenario test generation templates, see *Templates for Scenario Test Generation* on page 11.

Variables Used in Both the COM/VB and EJB Templates

The following table lists the replacement variables that are used in both COM/VB and EJB templates for scenario test generation:

Variable	Description	Template
<DIAGRAM_NAME>	Rose diagram name	Com_scenario_project Com_scenario_basmain Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario
<QUALIFIED_DIAGRAM_NAME>	Fully-qualified diagram name including the package hierarchy containing the diagram. Example: Use Case View::MyPackage::MyDiagram The template processor typically removes embedded blanks and changes the double-colons to either backslashes, "\", or periods, ".", depending on the usage (Java package or file path).	Com_scenario_basmain Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario
<AUTHOR_NAME>	Login ID supplied when you connect to a test datastore project.	Com_scenario_basmain Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario
<GENERATION_DATE>	System time	Com_scenario_basmain Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario
<DATAPOOL_NAME>	System assigned name of a datapool for the diagram.	Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario
<VARIABLE_INITIALIZATIONS>	Generated code for initializing script program variables used in datapools and as parameters in operations.	Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario

Variable	Description	Template
<OPERATIONS>	Generated code to call all of the operations implied by the user's selection of test targets.	Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario
<RETURN_VARIABLE>	If the operation returns a value, then this is a variable name generated by concatenating "retval_" with the Operation Name. May or may not be present in generated code.	Com_scenario_operation Scenario_java_method1
<CLASS_NAME>	Name of the Rose interface class.	Com_scenario_constructor Com_scenario_operation Scenario_java_method1 Scenario_java_method2 Weblogic_scenario_constructor Websphere_scenario_constructor sunj2ee_scenario_constructor
<VERIFICATION>	Generated code for handing a verification point inserted at the user's option by the code generator.	Com_scenario_operation Com_scenario_script Scenario_java_method1 Scenario_java_method2 Weblogic_scenario Websphere_scenario sunj2ee_scenario
<DIAGRAM_NAME>	Rose diagram name	Com_scenario_project Com_scenario_basmain Com_scenario_script Weblogic_scenario Websphere_scenario sunj2ee_scenario
<QUALIFIED_DIAGRAM_NAME>	Fully-qualified diagram name including the package hierarchy containing the diagram. Example: Use Case View::MyPackage::MyDiagram The template processor typically removes embedded blanks and changes the double-colons to either backslashes, "\", or periods, ".", depending on the usage (Java package or file path).	Com_scenario_basmain Com_scenario_script Weblogic_scenario Websphere_scenario Sunj2ee_scenario

Variables Used Only in the COM/VB Scenario Test Templates

The following table lists the replacement variables that are used only in the COM/VB templates for scenario test generation:

Variable	Description	Template
<VARIABLE_DECLARATIONS>	Generated code for declaring script program variables used in datapools and as parameters in operations.	Com_scenario_script
<SCRIPT_DIRECTORY>	The directory path where this script lives, relative to directory "root" where RQA scripts are stored. Example: \scenariotests\ratlbankacct\usecaseview	Com_scenario_script
<MODEL_NAME>	The name of the model with the file path and file extension removed. For example, given "C:\TEMP\mymodel.mdl", the <MODEL_NAME> is "mymodel".	Com_scenario_basmain
<INDEX>	If used, this is a monotonically increasing integer used to uniquely identify instances of like-named things. Typically not used in code generated for scenarios.	Com_scenario_basmain
<OPERATION_NAME>	Name of an operation. This corresponds to the Rose operation name.	Com_scenario_operation
<OPERATION_ARGLIST>	The argument list for an operation. Derived from the Rose parameters collection associated with an operation.	Com_scenario_operation
<COCLASS_NAME>	Name of the coclass implementing the Interface object referenced in the diagram.	Com_scenario_constructor
<COCLASS_GUID>	CoClass GUID that implements the interface that the operation belongs to. If more than one coclass implements the interface, the user is allowed to select the coclass to use.	Com_scenario_constructor

Variable	Description	Template
<COCLASS_PROGID>	Coclass_progID, or programmatic identifier, is a registry entry that is associated with a CLSID. Like the CLSID, the coclass_progID identifies a class, but with less precision. Example: Rose Operation := "Logical View::COM::MyComponent::_Account::Post" Rose Interface := "Logical View::COM::MyComponent::_Account" Rose CoClass := "Logical View::COM::MyComponent::Account" Before: Set obj = CreateObject("<prog_id>") After: Set obj = CreateObject("MyComponent.Account")	Com_scenario_constructor
<LIBRARY_NAME>	Name of the COM Library associated with the Rose Interface Class associated with the operation.	Com_scenario_constructor
<OBJECT_NAME>	Variable that gets populated with the name of the object instance (target object) in the interaction diagram. If the name of the target object is not present (it is not required), then the variable is populated with a concatenation of the COM library name and the coclass name.	Com_scenario_constructor

Variables Used Only in the EJB Scenario Test Templates

The following table lists the replacement variables that are used only in the EJB templates for scenario test generation:

Variable	Description	Templates Used In
<OPERATION_SIGNATURE>	In Java/EJB this is the entire signature of an operation call including the concatenation of the operation name and its argument list.	Scenario_java_method1 Scenario_java_method2
<HOME_INTERFACE_NAME>	The name of the home interface.	Weblogic_scenario_constructor Websphere_scenario_constructor sunj2ee_scenario_constructor

Replacement Variables for Stub Generation Templates

The replacement variables for the stub generation templates can be logically divided into the following categories:

- Variables used only in Visual Basic and COM
- Variables used only with Enterprise JavaBeans (EJBs)

For a list of the stub generation templates, see *Templates for Stub Generation* on page 19.

Replacement Variables for the COM/VB Stub Templates

The following table lists the replacement variables that are used in the COM/VB templates for stub generation:

Variable	Description	Template
<<!VBClassName!>>	Class name for the stub. Example: Account	VBCOMClass FunctionBody PropertyGetBody PropertyLetBody SubBody.template
<<!MethodBodyTemplate!>>	Causes the stub generator to use one of the seven method body templates to generate the declaration and definition for each public method in the stub. For more information, see <i>Templates for Stub Generation</i> on page 19.	VBCOMClass
<<!MethodName!>>	Name of the method. Example: Post	FunctionBody FunctionBodyWith outLookUp PropertyGetBody PropertyGetBodyW ithoutLookUp PropertyLetBody SubBody SubBodyWithoutLo okUp
<<!ParameterDeclarations!>>	Comma-separated list of parameter types and names to be included as part of the method declaration. Example: ByVal lAccountNo As Long, ByVal lAmount As Long	FunctionBody PropertyGetBody PropertyLetBody SubBody

Variable	Description	Template
<<!ReturnType!>>	Return type for the method. Example: String	FunctionBody FunctionBodyWithoutLookUp PropertyGetBody PropertyGetBodyWithoutLookUp
<<!ParameterNamesAsStrings!>>	Comma-separated list of parameter names as strings. Example: "lAccountNo", "lAmount"	FunctionBody PropertyGetBody PropertyLetBody SubBody
<<!ParameterNames!>>	Comma-separated list of parameter names. Example: lAccountNo, lAmount	FunctionBody PropertyGetBody PropertyLetBody SubBody
<<!ParameterValuesAsStrings!>>	Comma-separated list of parameter values converted to strings. Example: "CStr(lAccountNo), CStr(lAmount)" NOTE: For complex data types such as arrays, the generated code will need to be modified to compile and work.	FunctionBody PropertyGetBody PropertyLetBody SubBody
<<!LookUpTableName!>>	Name of the lookup table to be used for the method. Example: _Account_Post_L	FunctionBody PropertyGetBody PropertyLetBody SubBody
<<!ParameterValuesAsSumofStrings!>>	Parameter values converted to String and concatenated. NOTE: For complex data types such as arrays, the generated code will need to be modified to compile and work. Example: CStr(lAccountNo) + ", " + CStr(lAmount)	FunctionBody PropertyGetBody PropertyLetBody SubBody
<<!ReasonForNoLookupCodeGeneration!>>	Describes why lookup code was not generated. Example: This Method does not have parameters. Lookup code cannot be generated. Insert code here to add logic.	FunctionBodyWithoutLookUp PropertyGetBodyWithoutLookUp SubBodyWithoutLookUp

Replacement Variables for the EJB Stub Templates

The following table lists the replacement variables that are used in the EJB templates for stub generation:

Variable	Description	Template
<<!JavaPackage!>>	Package name of the EJB. Example: RationalBankAcct	Session_Home Session_Remote Session_Bean
<<!HomeInterfaceName!>>	Class name for the home interface. Example: ManageAccountsHome	Session_Home
<<!HomeMethods!>>	Method declarations for methods in the home interface. For example, RationalBankAcct.ManageAccounts create() throws javax.ejb.CreateException,java.rmi.RemoteException.	Session_Home
<<!RemoteInterfaceName!>>	Class name for the remote interface. Example: ManageAccounts	Session_Remote MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue.
<<!RemoteMethods!>>	Method declarations for methods in the remote interface. Example: void deleteAllCheckingAccts() throws java.rmi.RemoteException; void deleteAllSavingsAccts() throws java.rmi.RemoteException; ... java.math.BigDecimal withdrawFromChecking(long accountID, java.math.BigDecimal amount) throws java.rmi.RemoteException,javax.naming.NamingException,javax.ejb.EJBException; java.math.BigDecimal withdrawFromSavings(long accountID, java.math.BigDecimal amount) throws java.rmi.RemoteException,javax.naming.NamingException,javax.ejb.EJBException;	Session_Remote

Variable	Description	Template
<<!ImplementationClassName!>>	Class name of the EJB implementation class. Example: ManageAccountsBean	Session_Bean
<<!MethodBodyTemplate!>>	This variable causes the stub generator to use one of the four methodbody templates to generate the declaration and definition for each public method in the implementation class. The method body templates are chosen via the following criteria: MethodBodyWithoutLookUp – Used when lookup code cannot be generated either because the method has no parameters, or method has neither a return value nor exceptions, or one of the parameters is complex and lookup code cannot be generated automatically. MethodBodyWithoutExceptions – Used when the method does not throw exceptions. MethodBodyWithoutReturnValue – Used when method does not have a return value (for example, returns void). MethodBody – Used for all other methods.	Session_Bean
<<!ReturnType!>>	Return type for the method Example: java.math.BigDecimal	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutLookUp
<<!MethodName!>>	The Method Name Example: depositToChecking	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutLookUp MethodBodyWithoutReturnValue
<<!ParameterDeclarations!>>	Comma-separated list of parameter types and names to be included as part of the method declaration. Example: long accountID, java.math.BigDecimal amount	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutLookUp MethodBodyWithoutReturnValue

Variable	Description	Template
<<!Exceptions!>>	Comma-separated list of exceptions that the method throws to be included as part of the method declaration. Includes the word "throws". Example: throws java.rmi.RemoteException,javax.naming.NamingException,javax.ejb.EJBException	MethodBody MethodBodyWithoutLookUp MethodBodyWithoutReturnValue
<<!ParameterCount!>>	Number of parameter for the method Example: 3	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue
<<!ParameterNamesAsStringArray!>>	Code to turn parameter names into array of strings. Example: ParamNames[0] = "accountID"; ParamNames[1] = "amount";	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue
<<!Return!>>	The word "Return" + a space Example: Return	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutLookUp
<<!ParameterNames!>>	Comma-separated list of parameter names, for example accountID, amount	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue
<<!InitialReturnValue!>>	Initial value assigned to the return value—for example null. Zero for numeric types false for boolean null for String and Objects	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutLookUp
<<!LookupTableName!>>	Name of the lookup table to be used for the method—for example ManageAccountsBean_depositToChecking_L	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue
<<!ParameterValuesAsStringArray!>>	Code to turn parameter values into array of strings. For example: values[0] = Long.toString(accountID); values[1] = amount.toString();	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue

Variable	Description	Template
<<!ParameterValuesAsSumofStrings!>>	Parameter values are converted to sum of strings. Example: values[0] + " " + values[1] + " "	MethodBody MethodBodyWithoutExceptions MethodBodyWithoutReturnValue
<<!ConvertStringValueToReturnType!>>	Converts the expected return value read from the lookup table to the return value type. Example: retval = new java.math.BigDecimal(sRetval);	MethodBody MethodBodyWithoutExceptions
<<!CatchBlocks!>>	Catch block for exception that the method throws. It is used to throw the exception that lookup table instructs. Example: catch (java.rmi.RemoteException e) { throw e; } catch (javax.naming.NamingException e) { throw e; }	MethodBody MethodBodyWithoutReturnValue
<<!ReasonForNoLookupCodeGeneration!>>	Describes why lookup code was not generated. For example: //This Method does not have parameters. Lookup code cannot be generated. //Insert code here to add logic.	MethodBodyWithoutLookupUp

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

Resource File Not Found

Symptom

The following message appears when you open a Visual Basic project after generating a script:

```
File not found: FileName.Res
```

Explanation

During script generation, QualityArchitect compiles the resources for datapool configuration using the Microsoft Visual Studio resource compiler—*rc.exe*. The end result is a resource file—*test.res*.

If this fails for any reason, the original source file, *test.rc*, is left in the directory with the script. However, you need to specify where the *rc.exe* resides using the RQA Options dialog box. For more information, see *Requirements for Testing COM Components* on page 88.

EJB Class or Interface Not Found Messages

The following messages indicate that there are missing dependency relationships in the model between the bean implementation class, the home interface, the remote interface, and the primary key class. If you have the Rose J addin enabled, reverse engineering an EJB or creating a new EJB will create the necessary dependency relationships automatically. To enable script generation, be sure to insert the dependency relationships in your class diagram.

The messages are as follows:

- EJB remote interface not found
- EJB entity bean implementation class not found
- EJB home interface not found
- EJB session bean implementation class not found
- EJB primary key class not found
- Cannot find home, remote or key interface from implementation class. Check possible missing relationship in model.
- Cannot find implementation class for home, remote or key interface class. Check possible missing relationship in model.

Glossary

application-under-test. The software being tested. See also *system-under-test*.

baseline results. A persistent snapshot of data that is assumed to be correct and is used as the expected data object in a static verification point.

bean class. Class that actually implements the bean's business methods.

black-box testing. Tests that rely on a requirements definition or functional description of the application-under-test. A record and playback tool, such as Robot, is an example of a black-box testing tool.

class invariant. A set of rules that hold true when an object is in a stable state. The class invariant consists of a list of requirements on the data in a class. See also *Design by Contract*, *precondition*, and *postcondition*.

collaboration diagram. An interaction diagram that shows the sequence of messages that implement an operation or a transaction. Collaboration diagrams and sequence diagrams are alternate representations of an interaction. Collaboration diagrams show objects, their links, and their messages. They can also contain simple class instances and class utility instances. Each collaboration diagram provides a view of the interactions or structural relationships that occur between objects and object-like entities in the current model.

container-managed persistence. With container-managed persistence, the container is responsible for synchronizing the fields in an entity bean with data in a database.

context factory. A Java object that contains the JNDI information required by a naming service.

data-driven testing. A type of software testing that tests code with specific input and output values. With data-driven testing, test code is separated from test data. Data-driven tests are often used to compare expected and actual output values. See also *functional testing*.

database verification point. A type of verification point used with Rational Quality Architect. The database verification point tests the results of a SQL statement.

datapool. A source of test data that test scripts can draw from during playback. You can generate datapools using the Datapool Manager, or you can derive datapools from other sources such as your database.

Design By Contract. A programming methodology based on three important concepts: preconditions, postconditions, and a class invariant.

entity bean. An object representation of persistent data that is maintained in a permanent data store, such as a database. A primary key identifies each instance of an entity bean. Entity beans typically model business concepts that can be expressed as nouns, such as a customer or a bank account. Entity beans are transactional, and they are recoverable following a system crash.

functional test. A test to determine whether a system, application, or component functions as intended. Functional tests often compare how the application-under-test behaves in the current build against its behavior in previous builds.

Grid Comparator. The Rational Test component for reviewing, analyzing, and editing data files for text and numeric verification points in grid formats. The Grid Comparator displays the differences between the recorded baseline data and the actual data captured during playback.

home interface. An interface class for an enterprise bean that defines life-cycle methods for the bean—that is, methods for creating, removing, and finding beans.

IDE. Integrated Development Environment. This environment consists of a set of integrated tools that are used to develop a software application. Examples of IDEs supported by Rational Quality Architect include Visual Age and Visual Cafe.

interaction diagram. See *collaboration diagram* and *sequence diagram*.

log. A file that contains the record of events that occur during test script playback.

LogViewer. See Rational LogViewer.

playback. The process of executing a script.

postcondition. A requirement that a method, routine, or function has on its result. The requirement is checked after the method executes.

precondition. A requirement that a method, routine, or function has on the values supplied to its parameters. The requirement is checked before the method executes.

Rational Administrator. The Rational Test component that you use to create and maintain repositories, projects, users, groups, computers, and SQL Anywhere servers.

Rational LogViewer. The Rational Test component for displaying log files, which contain the record of events that occur while playing back a script or running a schedule. Also, the component from which you start the four Comparators.

Rational TestManager. The Rational Test product for managing the overall testing effort. You use it to define and store information about test documents, requirements, scripts, schedules, and sessions.

receiver object. The object that receives the messages in a Rose interaction diagram.

regression testing. Testing that occurs over successive builds of an application.

remote interface. An interface class for an enterprise bean that defines all of the bean's business methods.

requirements-based testing. A testing strategy based on test requirements. Requirements-based testing measures the number of test requirements that have been verified compared to all of the requirements that have been identified.

script. See *test script*.

sequence diagram. A graphical view of a scenario that shows object interaction in a time-based sequence—that is, what happens first, what happens next. Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces. This type of diagram is best used during early analysis phases in design because they are simple and easy to comprehend. Sequence diagrams are normally associated with use cases. Sequence diagrams are closely related to collaboration diagrams and both types of diagrams are alternate representations of an interaction. There are two main differences between sequence and collaboration diagrams: sequence diagrams show time-based object interaction while collaboration diagrams show how objects associate with one another.

session bean. A session bean typically models business concepts that can be expressed as a verb, such as managing an account or executing a transaction. A session bean is created by a client and in most cases exists only for the duration of a single client/server session. Session beans can be transactional, but (normally) they are not recoverable following a system crash. Session beans can be stateless or stateful.

stub. The minimum set of interfaces for a component that interacts with the component-under-test. Stubs can be used to control the results returned from components that are dependent on the component-under-test.

test case. A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

test script. Test scripts are the computer-readable instructions that automate the execution of a test procedure. Test scripts can be generated automatically using test automation tools, programmed using a programming language, or can result from a combination of recording, generating, and programming. With Rational test generation tools, the test script also includes a repository object. EJB test scripts, in particular, are generated in Java, and all generated Java test scripts extend the TestScript class.

test log. See log.

TestManager. See *Rational TestManager*.

verification point. A functional testing construct used by a test script to verify a specific behavior of the application or component-under-test. Static verification points capture information about the application-under-test and store it as the baseline. During playback, a verification point recaptures the object information and compares it to the baseline. Results of the verification point are maintained in a log, allowing for analysis of overall functional correctness and test case coverage.

white-box testing. Tests that rely on knowledge of the code, specifications, or other source material to perform the test. White-box testing is also called Structural testing. Rational QualityArchitect is an example of a white-box testing tool.

Index

A

- ActiveX Type Library 87
- actual data objects 23
- Administrator, starting 6
- application servers
 - editing hostName and portNumber 55

B

- bean class
 - definition 27

C

- classpath environment variable 39
- CloudScape database 32
- collaboration diagrams, testing 8
- COM components
 - generating unit tests for 90
 - requirements for testing 88
 - testing existing objects 87
- Connection page
 - Data Link Properties wizard 101
- correlation of data 11–16

D

- data correlation 11–16
- Data Link Properties wizard 100
- data types
 - list of standard data types 23
- database verification points 25
- datapools
 - correlation with message signatures in
 - Rose 14
 - creating 49, 93
 - definition 9, 21
 - sample code fragment 49

- datastores 6
- DB2 31
 - creating database for sample application 34
 - JDBC driver 60
 - JDBC driver for use with sample application 35
- directory for storing test scripts 6
- dynamic verification points 25

E

- editing
 - datapool data 94
- EJB
 - connecting with the Session Recorder 76
 - executing test scripts from TestManager 86
 - executing test scripts from VAJ 55
 - executing test scripts from Visual Cafe 56
 - generating unit tests 46
 - overview 27
 - Query Builder 59
 - requirements for testing 29
 - script generation overview 46
- EJB Session Recorder
 - generating test scripts from the XML log 86
 - inserting verification points 85
 - invoking a method on the home interface 79
 - starting 69
 - using with the sample application 75
- Enterprise JavaBeans. *See* EJB
- entity beans
 - in sample application 32
- expected data objects 23

F

- File not found error after generating COM/VB scripts 125

G

- generating
 - EJB unit test scripts 46
 - test scripts from the XML log 86

H

- home interface
 - and the Session Recorder 79
 - definition 27
 - invoking a method on 79

I

- implementation class
 - definition 27
- importing
 - type libraries 87
- Initial Context Factory 71, 74
 - definition 78
- Initial Data property in Rose 13
- interaction diagrams, testing 8

J

- Java Naming and Directory Interface. *See* JNDI
- JDBC
 - driver 57, 60
 - URL 57, 60, 61
- JNDI 76, 78
 - Provider URL 74
 - server 37

L

- load tests 10
- Location Server Daemon
 - for VAJ 37
- lookup tables
 - creating 18, 52, 95

M

- manual verification points 25
- measuring performance 10
- message signatures, in Rational Rose 11–16
- method test. *See* unit test
- MS Access
 - JDBC driver 60, 61

N

- Name and Type option, in Rose message
 - signatures 16
- Name-Only option, in Rose message
 - signatures 15
- Naming Service 77
- NoClassDefFoundError message 30

O

- OLE DB
 - Connection String 100, 102
 - Query Builder 99
 - Query Builder overview 99

P

- performance tests 10
- Persistent Name Server
 - for VAJ 37
- primary key class
 - definition 27
- projects, for storing test assets 6
- Provider page
 - Data Link Properties wizard 100
- Provider URL 77

Q

- Query Builder (Java) 59–66
- Query Design wizard
 - Java 62–64
 - OLE DB 103–104

R

- Rational Administrator, starting 6
- Rational Bank Account sample application
 - definition 32
 - deployment 36
 - importing into VAJ 33
 - picture of 33
- receiver objects, generating tests for 8
- regression tests 24
- remote interface
 - definition 27
- replacement variables
 - in RQA templates 107
- requirements, for using QualityArchitect 2
- reverse engineering 46
- Rose
 - ATL (ActiveX Type Library) object creation wizard 87
 - initial data property 13
 - type library import tool 87
- round-trip engineering 88, 89
- rqacomsample model 89
- rttseajava.jar file 30, 67
- rttssjava.jar file 30, 67

S

- sample EJB application
 - requirements 31
- scenario tests 8
 - datapools in 21
 - definition 6
 - for testing transactions 57
 - generating 57, 96
- sequence diagrams, testing 8
- session beans
 - in sample application 32
- Session Recorder
 - generating test scripts from the XML log 86
 - inserting verification points 85
 - invoking a method on the home interface 79
 - starting 69
 - using with the sample application 75
- signature options, in Rational Rose 12

SQL

- building queries 60, 62
- viewing query results 65
- SQL Server
 - JDBC driver 60
- standard data types
 - list of 23
- static verification points 24
- stubs
 - creating lookup tables for 52
 - definition 16
 - generating 18, 51, 94
 - importing into VAJ 53
 - templates for generating 19
- substitution variables
 - in RQA templates 107
- Sun J2EE Reference Server
 - deploying the sample application 42
- Swing
 - requirements for 30, 67
- Swing foundation class library
 - importing into VAJ 39
- swingall.jar file 30, 67

T

- techpubs x
- templates
 - for EJB stub generation 19, 51
 - for scenario test generation 11
 - for unit test generation 7
 - replacement variables in 107
- test assets 6
- test datastore 6
- Test Script Services
 - definition 21
- test scripts
 - executing from TestManager 56
 - executing from VAJ 55
 - executing from Visual Cafe 56
 - executing in Visual Basic 96
 - types of 5
- TestManager
 - executing test scripts 56

- TestScript class 28
- transaction test. *See* scenario test
- transactions
 - testing 57, 96
- TSSMeasure class 10
- type library import tool in Rational Rose 87
- Type-Only option, in Rose message
 - signatures 15

U

- unit tests
 - datapools in 22
 - definition 5
 - generating 46

V

- VAJ
 - deployment 36
 - executing test scripts 55
 - importing stubs 53
 - requirements 31
 - running the sample application 33
- VAJ EJB Development Environment
 - requirements 31
- variables
 - in RQA templates 107
- verification points
 - definition 8, 23
 - in scenario tests 57, 97
 - inserting with the Session Recorder 85
- Visual Age for Java. *See* VAJ

- Visual Basic
 - executing test scripts 96
 - programming in 88
 - project file 92
- Visual Cafe
 - configuration for sample application 43
 - executing test scripts 56
 - requirements 31

W

- WebLogic
 - default context factory 78
 - deployment 40
 - JDBC driver 60
 - Naming Service, URL for 77
 - requirements 31
- WebSphere
 - default context factory 78
 - deployment 36
 - JDBC driver 60
 - Naming Service, URL for 77
 - Test Environment Control Center 37
 - Test Environment requirements 31

X

- XML
 - Log File Path 76
- XML log
 - generating test scripts from 86
 - viewing 86