Rational the e-development company™

Rational Test Script Services for Java

VERSION 2001A.04.00

PART NUMBER 800-024529-000

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

Rational
the e-development company™

Contents

Preface

About This Manual

This manual is a reference of the methods that you call to add a variety of testing services to your test scripts — services such as datapool, logging, monitoring, synchronization, and verification point capabilities, as well as stub services for testing EJB components.

The Test Script Services described in this manual are designed to be used with Rational TestManager and Rational QualityArchitect.

Audience

This manual is intended for test designers who write or edit test scripts in Java. Your Java test scripts can be used for both performance and functional testing.

Other Resources

- To access an HTML version of this manual, click **TSS for Java** in the following default installation path (*ProductName* is the name of the Rational product you installed, such as Rational TestStudio):

  Start > Programs > Rational *ProductName* > Rational Test > API

- All manuals for this product are available online in PDF format. These manuals are on the *Rational Solutions for Windows* Online Documentation CD.
- For information about training opportunities, see the Rational University Web site: http://www.rational.com/university.

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

Note: 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)

Introduction 1

About Java Test Script Services

Rational *Test Script Services* are testing services that you can add to your Java test scripts through the methods described in this reference. For example, you can add logging, synchronization, timing, and datapool services to your test scripts. You can also add verification services to validate the behavior of Java components, such as Enterprise Java Beans (EJBs).

Test Script Services can be used with either or both of the following products:

- Rational TestManager
- Rational QualityArchitect

Note: This product does not include the Java Development Kit (JDK). You must ensure that a version of the JDK exists on the master or agent computers prior to playback.

Using Test Script Services With Rational TestManager

Rational TestManager is a product that lets you plan, design, implement, execute, and analyze tests, including system-level and component tests of functionality and performance.

TestManager fully supports Java test scripts enhanced with Test Script Services functionality — for example:

- TestManager will adhere to any synchronization and delay functionality in your script when it plays back (executes) the script within a suite of scripts.
- During script playback, a tester can monitor script runtime states through the script monitoring methods.
- TestManager reports display the results of timed operations.
- TestManager test cases can be associated with Java scripts that contain measurement inputs, such as verification methods for validating the behavior of objects.

- TestManager can run your Java scripts with scripts of other types, such as Visual Basic, GUI, and VU scripts.

The Test Script Services used with TestManager are documented in *Test Script Services Reference* on page 17.

Using Test Script Services With Rational QualityArchitect

Rational QualityArchitect is a product that lets you test objects such as EJB and COM/DCOM components. You can test, or *verify*, the behavior of EJB components using the verification services documented in the following sections:

- *Verification Services* on page 119
- *Database Verification Point Reference* on page 131
- *Implementing a New Verification Point* on page 161
- *Verification Point Framework Reference* on page 181

Note: This document is primarily a reference document. For information on how to use Rational QualityArchitect, see the *Using Rational QualityArchitect* manual.

Summary of Services

The following table describes the categories of Test Script Services that are available with TestManager and QualityArchitect. It also specifies the product(s) that the categories of services are commonly used with.

Category	**Description**	**Commonly Used With**
Datapool	Provide variable data to test scripts during playback, allowing virtual testers to send different data to the server with each transaction.	TestManager, QualityArchitect
Logging	Log messages for reporting and analysis.	TestManager, QualityArchitect
Measurement	Provide the means of fine tuning and controlling your tests through operations such as timing actions, setting think time delays, and setting environment variables.	TestManager
Utility	Perform common test script operations such as retrieving error information, controlling the generation of random numbers, and printing messages.	TestManager, QualityArchitect

Category	Description	Commonly Used With
Monitor	Monitor test script playback progress.	TestManager, QualityArchitect
Synchronization	Synchronize multiple virtual testers running on a single computer or across multiple computers.	TestManager
Session	Manage test script session execution and playback.	TestManager, QualityArchitect
Advanced	Perform advanced logging and timing operations.	TestManager
Verification Point	Validate the behavior of objects such as EJB components.	QualityArchitect

As indicated at the end of the preface, an HTML version of this manual is available from the **Start** menu and a PDF version from the Rational documentation CD.

Working with Test Scripts 2

About Java Test Scripts

A Java *test script* is a Java source file used for testing applications and components within the Rational test environment.

Java test scripts can be used in functional, performance, and component testing, and they typically include calls to Test Script Services. Compiled Java test scripts can be run either standalone or within a TestManager suite.

You work with test scripts by using both TestManager and your Java IDE, as described in this chapter.

Creating Test Scripts

You can create a Java test script in any of these ways:

- Generate a script with the Rational QualityArchitect Session Recorder.

  The Session Recorder records your interactions with the EJB object you are testing, and then automatically generates a script that can reproduce your actions exactly as you recorded them.

- Generate a script from a Rational Rose model. If you create test scripts by this method, you can begin testing components that are still in the design stage and not yet fully implemented.

  This type of script generation requires both Rational Rose and Rational QualityArchitect.

- Manually write a Java script using a supported Java IDE. See the Release Notes for supported IDEs.

  If you are not using Rational QualityArchitect and Rational Rose, this is the only way to produce a Java test script.

Entry Points

A Java test script must extend the following base class:

```
com.rational.test.tss.TestScript
```

The entry point that you need to include in your test scripts varies, depending on whether you intend to run the script inside or outside of TestManager. For more information, see *Running Test Scripts* on page 8.

Registering Test Script Source Folders

If you create a test script in your IDE and manually code it, you must inform TestManager of the root *test script source folder* where the script is stored. To do so:

1. Click **Tools > Manage > Test Script Types**.
2. Select **Java Script**, and then click **Edit**.
3. Click the **Sources** tab, and then click **Insert**.
4. On the **General** tab, type a name for the test script source folder.

   This name will be added to TestManager's **File** menus. for You select this name when creating, editing, and running test scripts stored in the source folder.
5. Click the **Connect Info** tab, and then type the full path of the test script source folder in the **Data path** boxThis will be the name of a project folder that you have created using your IDE.
6. Type the following values, exactly as shown, into the **Option name** and **Option value** columns:

Option name	Option value
Type	Java
Default datastore	0

7. Click **OK**. The new source folder name appears in the **Sources** list.

Editing and Storing Test Scripts

All of your test script editing is done inside of your IDE. You can open a test script directly from your IDE or from TestManager.

To open a test script in TestManager, click **File > Open Test Script** > *type*, where *type* is the Java source folder that you created in section *Registering Test Script Source Folders* on page 6. Then select the script you want to open. TestManager checks the Windows Registry to find the IDE associated with the test script. If TestManager doesn't find an IDE associated with the test script, it opens Windows Notepad.

When you save a test script, you must store the script outside of any Rational projects and datastores. You store the scripts in a Java test script source folder that you create.

You specify the location of the test script source folder in different ways, depending on how you created the script:

- If you manually code a test script and you have not yet created a test script source folder for the current project, do the following:
  1. Create the folder where you want to store the test script source file.
  2. Register the test script source folder using the instructions in the section *Registering Test Script Source Folders* on page 6.
- If you auto-generate a script with Rational QualityArchitect (using the Session Recorder or through a Rose model) and you have not yet created a test script source folder for the current project, do the following:
  1. Create the folder where you want to store the test script source file.
  2. At script-generation time, you are prompted to specify the folder where you want to store the test script being generated. Be sure to select a location that everyone on the project can access.

     Note: When specifying the folder, use a Universal Naming Convention (UNC) path — for example: \\server-name\directory-path.

     Any future scripts that you create for this project are stored in the same test script source folder. This location cannot be changed once it is defined.

Only script files are stored outside of the Rational project. TestManager stores other related files, such as any datapool and log files, as well as references to the script files, within the current Rational project.

For information about how TestManager stores compiled scripts at test runtime, see *Returning Information from Test Scripts* on page 12.

Storing Scripts in Java Packages

If a script is part of a Java package, the script must be stored in a path that consists of the test script source folder path plus the name of the package. For example, if the path is D:\TestScripts and the test script you are storing is included in the package com.rational.test, store the script in the following location:

D:\TestScripts\com\rational\test

Test Script Names

Java test script names follow standard Java naming conventions.

The maximum name length of scripts stored outside of the Rational datastore is limited only by the constraints of the operating system.

Compiling Test Scripts

When running a test script, TestManager checks the timestamp of the compiled script. If the compiled script is out of date, TestManager compiles the script before running it.

To compile a script, TestManager locates the compiler javac.exe on your computer's system path. If TestManager can't find a compiler, it generates an error.

For information about running scripts with TestManager, see the *Using Rational TestManager* manual.

A Java script is compiled to a .class file. By default, the file is stored in the test script source folder.

The .class file is assigned the same root name as the .java file.

If a script contains inner classes (classes declared within classes), each class is compiled to its own file.

Running Test Scripts

You can run test scripts either from within or outside of TestManager. Test scripts that you execute from within TestManager can run on the local host or on an agent host.

Where you run a test script depends, in part, upon your reason for running it:

- To run a test. With TestManager, you can run a single test script by itself (**File > Run Test Script**), from within a test case (**File > Run Test Case**), or you can add the script to a TestManager suite and run the suite.

Performance tests are typically run within TestManager. Component tests conducted with QualityArchitect can be run either within TestManager or your IDE.

- To debug a test script. If you are debugging a test script, run the script from your IDE rather than from TestManager.

In order to run a test script from TestManager that was generated by QualityArchitect, you must include in your CLASSPATH the full paths for:

- Any client .jar files referenced by EJBs, either in the recording or in the component-under-test
- When testing in an environment such as WebLogic or WebSphere, any .jar files required by the Application Server
- For generated scripts that include verification points, the JavaHelp (jh.jar) file

For other test scripts containing only Rational classes that are run from TestManager, you do not need to modify your CLASSPATH. This is true whether the test script executes on the local host or on an agent. You do not have to copy any files to the agent or modify its CLASSPATH.

For test scripts containing Rational classes that are run outside TestManager, their full pathnames must be specified in your CLASSPATH. The following table lists the relevant .jar files, their default paths, and product(s) that use them.

File	Installed Location	Required for
rational_ct.jar	Rational Test\QualityArchitect	QualityArchitect
rttseajava.jar	Rational Test\tsea	QualityArchitect TestManager
rttssjava.jar	Rational Test	QualityArchitect TestManager
swingall.jar	Rational Test\QualityArchitect	QualityArchitect

For test scripts containing private classes (classes that are unknown to TestManager or QualityArchitect), the full pathnames of these must be specified in your CLASSPATH. This is true whether the test scripts execute within or outside TestManager. In addition, for test scripts executed from TestManager that run on an agent, the .jar files must be present on the agent, and their full paths must be specified in the agent's CLASSPATH.

Running Test Scripts in a TestManager Suite

A TestManager *suite* is a collection of test scripts. In TestManager, you typically run tests by running a single script or a number of scripts in a suite.

You can combine scripts of different types in the same suite — for example, you can add your Java scripts to a suite that also contains Visual Basic, GUI, and VU scripts, and even scripts of a custom test type.

For information about adding scripts to a TestManager suite, see the *Using Rational TestManager* manual.

A .java test script that you want to run inside a suite must implement the `testMain()` method. This method is the entry point for the class.

The following is an example of a skeletal .java test script that includes the `testMain()` method, shown in bold type, that TestManager needs in order to run the script:

```
import java.io.*;
import com.rational.test.tss.*;

public class Hello extends com.rational.test.tss.TestScript {

   public void testMain(String[] args) {

   // Your test script code goes here
   }
}
```

Adding a Source Folder for Java Scripts

For TestManager to run a Java script, you must create a Java test script source folder (if one doesn't already exist for the current folder) and place the script into it. At suite runtime, TestManager compiles the script, places the resulting .class file in the same folder, and then executes the .class file.

For example, suppose you want to manually create and code a script named Script1.java and run it from a folder named D:\TestScripts, which doesn't exist. You would do the following:

1 Create the folder D:\TestScripts.

2 Create the script Script1.java and save it to D:\TestScripts.

3 Register the test script source folder D:\TestScripts with TestManager. For information, see *Registering Test Script Source Folders* on page 6.

4 Add Script1.java to a TestManager suite and run the suite.

When the suite is run, TestManager compiles Script1.java, places the resulting .class file in D:\TestScripts, and executes Script1.

Adding a Script Contained in a Java Package

In Java, a *package* lets you assign a single name to a group of related classes.

If you want TestManager to run a Java test script that is part of a package, the source and the .class runtime must both be located in an appropriate folder below the test script source folder. The folder's path name is determined by the name of the test script source folder plus the name of the package.

For example, suppose you want to manually create and run a script named Script1, which is located in the package com.rational.test. You want to run the script from a folder named D:\TestScripts, which doesn't exist. You would do the following:

1 Create the folder D:\TestScripts.

2 Create the folders \com\rational\test below the test script source folder D:\TestScripts.

3 Place the script Script1.java in D:\TestScripts\com\rational\test.

4 Register the test script source folder D:\TestScripts with TestManager. For information, see *Registering Test Script Source Folders* on page 6.

5 Add Script1.java to a TestManager suite and run the suite.

When you run the suite, TestManager compiles Script1.java, places the resulting .class file in D:\TestScripts\com\rational\test, and executes Script1 using the class name com.rational.test.Script1.

Running Test Scripts Outside TestManager

A test script that you want to run from your IDE must include a `main()` entry point as well as a `testMain()` entry point.

The following example extends the previous example on page 10 by including the code, shown in bold type, required for running and debugging the script in your IDE:

```
import java.io.*;
import com.rational.test.tss.*;

public class Hello extends com.rational.test.tss.TestScript {

   public static void main(String[] args) {
      Hello h = new Hello();
      h.testMain(args);
   }

   public void testMain(String[] args) {
```

```
   // Your test script code goes here
   }
}
```

The following example further extends the skeletal test script shown above. This example illustrates the inclusion of Test Script Services calls and the creation of a debug file usable from your IDE.

```
import java.io.*;
import com.rational.test.tss.*;

public class Hello extends com.rational.test.tss.TestScript {
   public static void main(String[] args) {
      Hello h = new Hello();
      h.testMain(args);
   }
   public void testMain(String[] args) {
      try {
         FileOutputStream debugfile = new
         FileOutputStream("Hello.dat",true);
         PrintStream deb = new PrintStream(debugFile);
         deb.println("Hello World");
         System.out.println("Hello World");
         System.out.println("Starting first sleep for 5 seconds");
         TSSMeasure.commandStart("string1", "string1", 0);
         Thread.sleep(5000);
         com.rational.test.tss.TSSNamedValue[] a = null;
         TSSMeasure.commandEnd((short) 0,"string1",0,0,"string2",a);
         TSSMeasure.think();
         System.out.println("Starting first sleep for 1 second");
         Thread.sleep(1000);
         System.out.println("Hello World done");
         }
   }
}
```

Returning Information from Test Scripts

Test Script Services calls can deposit information in any of these locations:

- Test log
- Error and output files
- TestManager shared memory

The following sections describe these locations.

Test Log

TestManager uses the test log (or *log*) to list the test cases that have been run and record whether they pass or fail. TestManager generates reports based on the logged information.

You can also write pass/fail results to the log as well as log messages and report errors.

The following are the Test Script Services logging methods:

- *TSSLog.event()* on page 33
- *TSSLog.message()* on page 34
- *TSSLog.testCaseResult()* on page 36
- *TSSMeasure.commandEnd()* on page 38
- *TSSMeasure.commandStart()* on page 40
- *TSSAdvanced.logCommand()* on page 99
- *TestLog.writeException()* on page 115
- *TestLog.writeStubException()* on page 116
- *TestLog.writeStubMessage()* on page 117

For additional information about logging exceptions, see *Catching Exceptions* on page 15.

TestManager determines the location of the log file as follows:

- If the test script is running within TestManager, or if it is running outside of TestManager but against a TSS Server through rttssee.exe, the location is determined by the parent process, not by the test script.
- If the test script is a Rational QualityArchitect test script running in the IDE, the location is again determined by the parent process.
- If the test script is running outside TestManager and the TSS Server is not running, the location, by default, is relative to the current directory and is referenced as ./u000. Use `TSSSession.context()` to control the location of the log file.

Error File and Output File

As a development and debugging aid, you can write information to an error file and an output file.

Use the utility methods `stdErrPrint()` and `stdOutPrint()` to write to the error and output files.

TestManager determines the location of the error and output files as follows:

- If the test script is running within TestManager, the location is determined by the parent process, not by the test script.
- If the test script is running outside TestManager but against a TSS Server through rttssee.exe, the location is determined by command-line options you set:
  - With no command-line options used, the error file is the system standard error file, and the output file is the system standard output file.
  - With the -r option, the error and output files are stored in the working directory. The working directory is the system's current working directory, unless a different location is specified through the -d option.

    Set the error file name with e<usernumber> and the output file name with o<usernumber>. The variable <usernumber> defaults to 0 and is set by the -u command-line option.
- If the test script is running outside TestManager and the TSS Server is not running, the error file is the system standard error file, and the output file is the system standard output file.

TestManager Shared Memory

Shared memory is used to provide data for TestManager's runtime console. Shared memory is also used to pass information between test scripts.

To write data to shared memory, use the methods described in the following sections:

- *Monitor Class* on page 72. Use the `TSSMonitor` methods to provide data that is used during TestManager's monitoring operations.
- *Synchronization Class* on page 82. Use the `TSSSync` methods to allow concurrently running scripts to share data.

These methods work only in test scripts that are run from TestManager.

Catching Exceptions

If you catch exceptions in your test script, you are intercepting the exceptions before TestManager can become aware of them. If you handle the exception and take no other action, the script continues to run, and TestManager could log a Pass result for the script.

If an exception occurs and the script does not contain exception handling logic, the test script stops running, the next script in the suite is run, and TestManager logs a Fail result for the script and a description of the exception.

If you want to catch certain exceptions, but you want the log to reflect a Fail result for the test script, use one of the Test Script Services logging methods to log the Fail result.

Alternatively, consider catching the exception, logging an informative error message (that says, for instance, what you were trying to do in the script when the exception was thrown), and then re-throwing the exception to pop out of the script.

The following is an example of a catch block that re-throws an exception:

```
catch(Exception e {
   System.err.println("Exception handled in method");

   throw e;              // Re-throw for further processing

}
```

Test Script Services Reference 3

About Test Script Services

This chapter describes the Rational Test Script Services (TSS). It explains the methods you use to give test scripts access to services such as datapools, measurement, virtual tester synchronization, and monitoring. The methods are divided into the following functional categories.

Category	**Description**
Datapool	Provide variable data to test scripts during playback.
Logging	Log messages for reporting and analysis.
Measurement	Manage timers and test variables.
Utility	Perform common test script functions.
Monitor	Monitor test script playback progress.
Synchronization	Synchronize virtual testers in multi-computer runtime environments.
Session	Manage the test suite runtime environment.
Advanced	Perform advanced logging and measurement functions.

Datapool Class

During testing, it is often necessary to supply an application with a range of test data. Thus, in the functional test of a data entry component, you may want to try out the valid range of data, and also to test how the application responds to invalid data. Similarly, in a performance test of the same component, you may want to test storage and retrieval components in different combinations and under varying load conditions.

A *datapool* is a source of data stored in a Rational project that a test script can draw upon during playback, for the purpose of varying the test data. You create datapools from TestManager, by clicking **Tools > Manage > Datapools**. For more information, see the datapool chapter in the *Using Rational TestManager* manual. Optionally, you can import manually-created datapool information stored in flat ASCII Comma Separated Values (CSV) files, where a row is a newline-terminated line and columns are fields in the line separated by commas (or some other field-delimiting character).

Applicability

Commonly used with TestManager and QualityArchitect.

Summary

Use the datapool methods listed in the following table to access and manipulate datapools within your scripts. These are static methods of class `TSSDatapool`.

Method	Description
`close()`	Closes a datapool.
`columnCount()`	Returns the number of columns in a datapool.
`columnName()`	Returns the name of the specified datapool column.
`fetch()`	Moves the datapool cursor to the next row.
`open()`	Opens the named datapool and sets the row access order.
`rewind()`	Resets the datapool cursor to the beginning of the datapool access order.
`rowCount()`	Returns the number of rows in a datapool.

Method	Description
`search()`	Searches a datapool for the named column with a specified value.
`seek()`	Moves the datapool cursor forward.
`value()`	Retrieves the value of the specified datapool column.

TSSDatapool.close()

Closes a datapool.

Syntax–

```
int close()
```

Return Value

This exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.

Comments

Only one open datapool at a time is supported. A `close()` is thus required between intervening `Oopen()` calls. For a script that opens only one datapool, `close()` is optional.

Example

This example opens the datapool `custdata` with default row access and closes it.

```
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
int retVal = dp.close();
```

See Also

open()

TSSDatapool.columnCount()

Returns the number of columns in a datapool.

Syntax

```
int columnCount ()
```

Return Value

On success, this method returns the number of columns in the open datapool.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example opens the datapool `custdata` and gets the number of columns.

```
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
int columns = dp.columnCount();
```

TSSDatapool.columnName()

Gets the name of the specified datapool column.

Syntax

```
String columnName (int columnNumber)
```

Element	**Description**
`columnNumber`	A positive number indicating the number of the column whose name you want to retrieve. The first column is number 1.

Return Value

On success, this method returns the name of the specified datapool column.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier or column number is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example opens a three-column datapool and gets the name of the third column.

```
TSSDatapool dp = New TSSDatapool;
dp.open ("custdata");
if (dp.fetch())
   String colName = dp.columnName(3);
```

TSSDatapool.fetch()

Moves the datapool cursor to the next row.

Syntax

```
boolean fetch()
```

Return Value

This method returns `true` (success) or `false` (end-of-file).

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This call positions the datapool cursor on the next row and loads the row into memory. To access a column of data in the row, call `value()`.

The "next row" is determined by the *`assessFlags`* passed with the open call. The default is the next row in sequence. See `open()`.

After a datapool is opened, a `fetch()` is required before the initial row can be accessed.

An end-of-file (`TSS_EOF)` condition results if a script fetches past the end of the datapool, which can occur only if access flag `TSS_DP_NOWRAP` was set on the open call. If the end-of-file condition occurs, the next call to `value()` throws an exception.

Example

This example opens datapool `custdata` with default (sequential) access and positions the curson to the first row.

```
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
boolean retVal = dp.fetch();
```

See Also

`open(),seek(),value()`

TSSDatapool.open()

Opens the named datapool and sets the row access order.

Syntax

```
void open(String name, int accessFlags, TSSNamedValue[]
    overrides)

void open(String name)
```

Element	Description
name	The name of the datapool to open. If *accessFlags* includes `TSS_DP_NO_OPEN`, no CSV datapool is opened; instead, *name* will refer to the contents of *overrides* specifying a one-row table. Otherwise, the CSV file *name* in the Rational project is opened.
accessFlags	Optional flags indicating how the datapool is accessed when a script is played back. Specify at most one value from each of the following categories: **1** Specify the sequence in which datapool rows are accessed: `TSS_DP_SEQUENTIAL` – physical order (default) `TSS_DP_RANDOM` – any order, including multiple access or no access `TSS_DP_SHUFFLE` – access order is shuffled after each access **2** Specify what happens after the last datapool row is accessed: `TSS_DP_NOWRAP` – end access to the datapool (default) `TSS_DP_WRAP` – go back to the beginning **3** Specify whether the datapool cursor is shared by all virtual testers or is unique to each: `TSS_DP_SHARED` – all virtual testers work from the same access order (default) `TSS_DP_PRIVATE` – virtual testers each work from their own sequential, random, or shuffle access order **4** `TSS_DP_PERSIST` specifies that the datapool cursor is persistent across multiple script runs. For example, with a persistent cursor, if the row number after a suite run is 100, the first row accessed in a subsequent run will be numbered 101. Not valid with `TSS_DP_RANDOM` or `TSS_DP_PRIVATE`. **5** `TSS_DP_REWIND specifies that the datapool should be rewound when opened. Can be used only with TSS_DP_PRIVATE.` **6** `TSS_DP_NO_OPEN` specifies that, instead of a CSV file, the opened datapool will consist only of column/value pairs specified in a local array *overrides*[].
overrides	A local, two-dimensional array of column/value pairs, where *overrides*[n].*name* is the column name and *overrides*[n].*value* is the value returned by `value()` for that column name. See *TSSNamedValue* on page 217 for the implementation of this argument's data type.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The *`accessFlags`* are or result in an invalid combination.
- `TSS_NOTFOUND`. No datapool of the given *`name`* was found.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

If *`accessFlags`* are specified as 0 or are omitted, the rows are accessed in the default order: sequentially, with no wrapping, and with a shared cursor. If multiple *`accessFlags`* are specified, they must be valid combinations as explained in the syntax table. Any*`accessFlags`* specified with `open()` override those specified with the datapool configuration statements (see the example section).

If you close and then reopen a private-access datapool with the same *`accessFlags`* and in the same or a subsequent script, access to the datapool is resumed as if it had never been closed.

A test script that will be executed by TestManager can open only one datapool at a time.

If multiple virtual testers access the same datapool in a suite, the datapool cursor is managed as follows:

- The first open that uses the `TSS_DP_SHARED` option initializes the cursor. In the same suite run (and, with the `TSS_DP_PERSIST` flag, in subsequent suite runs), virtual testers that subsequently use the same datapool opened with `TSS_DP_SHARED` share the initialized cursor.
- The first open that uses the `TSS_DP_PRIVATE` option initializes the private cursor for a virtual tester. In the same suite run, a subsequent open that uses `TSS_DP_PRIVATE` sets the cursor to the last row accessed by that virtual tester.

An exception will be thrown if `open()` is called more than once (for a given instance of the class) without an intervening `close()` call. The exception message is "open was called without closing the previously opened Datapool". A call to `TSSException.getReturnValue()` in the catch block will identify the datapool that was already open when the call was made.

Example

This example opens the datapool named `custdata`. The datapool configuration statements, which may occur anywhere in the script, name the datapool and set the default row access.

```
TSSDatapool dp = new TSSDatapool();
dp.open("custdata");
...
public static class DatapoolConfig extends DatapoolInfo {
        public DatapoolConfig() {
            setDatapoolName("custdata");
            setDatapoolAccessFlags(TSS_DP_WRAP |
                                   TSS_DP_SEQUENTIAL |
                                   TSS_DP_SHARED);
        }
    }
```

See Also

`close()`

TSSDatapool.rewind()

Resets the datapool cursor to the beginning of the datapool access order.

Syntax

void **`rewind`**()

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The datapool is rewound as follows:

- For datapools opened `DP_SEQUENTIAL`, `rewind()` resets the cursor to the first record in the datapool file.
- For datapools opened `DP_RANDOM` or `DP_SHUFFLE`, `rewind()` restarts the random number sequence.
- For datapools opened `DP_SHARED`, `rewind()` has no effect.

At the start of a suite, datapool cursors always point to the first row.

If you rewind the datapool during a suite run, previously accessed rows are fetched again.

Example

This example opens the datapool `custdata` with default (sequential) access, moves the access to the second row, then resets access to the first row.

```
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
dp.seek(2);
dp.rewind();
```

TSSDatapool.rowCount()

Returns the number of rows in a datapool.

Syntax

```
int rowCount()
```

Return Value

On success, this method returns the number of rows in the open datapool.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example opens the datapool `custdata` and gets the number of rows in the datapool.

```
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
int rows = dp.rowCount();
```

TSSDatapool.search()

Searches a datapool for a named column with a specified value.

Syntax

```
void search (TSSNamedValue[] keys)
```

Element	Description
keys	An array containing values to be searched for. See *TSSNamedValue* on page 217 for the implementation of this argument's data type.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_EOF`. The end of the datapool was reached.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

When a row is found containing the specified values, the cursor is set to that row.

Example

This example searches the datapool `custdata` for a row containing the column named `Last` with the value `Doe`:

```
TSSNamedValue[] toFind = new TSSNamedValue[1];
toFind[0] = new TSSNamedValue();
toFind[0].name = "Last";
toFind[0].value = "Doe";
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
if (dp.fetch())
   dp.search(toFind);
```

TSSDatapool.seek()

Moves the datapool cursor forward.

Syntax

`void` **`seek`**`(int` *`count`*`)`

Element	Description
`count`	A positive number indicating the number of rows to move forward in the datapool.

Return Value

Exceptions

This method may throw an exception with one of the following values:

- `TSS_EOF`. The end of the datapool was reached.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The datapool identifier is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This call moves the datapool cursor forward *`count`* rows and loads that row into memory. To access a column of data in the row, call `value()`.

The meaning of "forward" depends on the *`accessFlags`* passed with the open call; see `open()`. This call is functionally equivalent to calling `fetch()` *`count`* times.

In addition to throwing an exception on error, this method returns a boolean status indicator where `false` indicates end-of-file (`TSS_EOF`). A script can check for this condition.

If a script fetches past the end of the datapool (as a result of `TSS_DP_NOWRAP` being set), the next call to `TSSDatapool.value()` will throw an exception.

Example

This example opens the datapool `custdata` with the default (sequential) access and moves the cursor forward two rows.

```
TSSDatapool dp = new TSSDatapool();
dp.open("custdata");
dp.seek(2);
```

See Also

`fetch(),open(),value()`

TSSDatapool.value()

Retrieves the value of the specified datapool column in the current row.

Syntax

DatapoolValue **`value`**(String *`columnName`*)

Element	**Description**
`columnName`	The name of the column whose value you want to retrieve.

Return Value

On success, this method returns the value of the specified datapool column in the current row.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_EOF`. The end of the datapool was reached.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The specified*`columnName`* is not a valid column in the datapool.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This call gets the value of the specified datapool column from the current datapool row, which will have been loaded into memory either by `fetch()` or `seek()`.

By default, the returned value will be a column from a CSV datapool file located in a Rational datastore. If the datapool open call included the `TSS_DP_NO_OPEN` access flag, the returned value will come from an override list provided with the open call.

Datapools store all data as strings. As a consequence, a retrieved value that is not really a string must be converted. To facilitate conversions, the class `DatapoolValue` wraps the value returned by `TSSDatapool.value()` and the following conversion methods are provided:

This method	**Generates**
`booleanValue()`	The `boolean` representation of the datapool value.
`byteValue()`	The byte representation of the datapool value.
`charValue()`	The character representation of the datapool value.
`floatValue()`	The `float` representation of the datapool value.
`getBigDecimal()`	The `BigDecimal` representation of the datapool value.
`intValue()`	The `int` representation of the datapool value.
`longValue()`	The `long` representation of the datapool value.
`shortValue()`	The `short` representation of the datapool value.
`toString()`	The `String` representation of the datapool value.

See *DatapoolValue* on page 218 for the implementation of this class.

Example

This example retrieves the value of the column named `Middle` in the first row of the datapool `custdata`.

```
TSSDatapool dp = new TSSDatapool();
dp.open("custdata");
if (dp.fetch()==true)
   phonebook.queryPerson(dp.value("Middle").toString());
   // queryPerson method expects a String parameter
```

See Also

`fetch(),open(),seek()`

Logging Class

Use the logging methods to build the log that TestManager uses for analysis and reporting. You can log events, messages, or test case results.

A logged event is the record of something that happened. Use the environment variable `EVAR_LogEvent_control` (page 44) to control whether or not an event is logged.

An event that gets logged may have associated data (either returned by the server or supplied with the call). Use the environment variable `EVAR_LogData_control` (page 44) to control whether or not any data associated with an event is logged.

Applicability

Commonly used with TestManager and QualityArchitect.

Summary

Use the methods listed in the following table to write to the TestManager log. They are static methods of class TSSLog.

Method	**Description**
`event()`	Logs an event.
`message()`	Logs a message event.
`testCaseResult()`	Logs a test case event.

TSSLog.event()

Logs an event.

Syntax

```
void event (String eventType,  short result, String
   description, TSSNamedValue[] property)
```

```
void event (String eventType,  short result)
```

Element	Description
eventType	Contains the description to be displayed in the log for this event.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `TSS_LOG_RESULT_NONE` (default: no notification) ▪ `TSS_LOG_RESULT_PASS` ▪ `TSS_LOG_RESULT_FAIL` ▪ `TSS_LOG_RESULT_WARN` ▪ `TSS_LOG_RESULT_STOPPED` ▪ `TSS_LOG_RESULT_INFO` ▪ `TSS_LOG_RESULT_COMPLETED` ▪ `TSS_LOG_RESULT_UNEVALUATED` `0` specifies the default.
description	Contains the string to be put in the entry's failure description field.
property	An array containing property name/value pairs, where `property[n].name` is the property name and `property[n].value` is its value. See *TSSNamedValue* on page 217 for the implementation of this argument's data type.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. An unknown *`result`* was specified.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The event and any data associated with it are logged only if the specified `result` preference matches associated settings in the `EVAR_LogData_control` (page 44) or `EVAR_LogEvent_control` (page 44) environment variables. Alternatively, the logging preference can be set with the `EVAR_Log_level` (page 45) and `EVAR_Record_level` (page 46) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

Example

This example logs the beginning of an event of type `Login Dialog`.

```
TSSNamedValue[] scriptProp = new TSSNamedValue[2];
scriptProp[0] = new TSSNamedValue();
scriptProp[0].name = "ScriptName";
scriptProp[0].value = "Login";
scriptProp[1] = new TSSNamedValue();
scriptProp[1].name = "LineNumber";
scriptProp[1].value = "1";
TSSLog.event("Login Dialog",0,"Login script failed",scriptProp);
```

TSSLog.message()

Logs a message.

Syntax

```
void message(String message,  short result, String description)
void message(String message)
```

Element	Description
message	Specifies the string to log.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ TSS_LOG_RESULT_NONE (default: no notification) ▪ TSS_LOG_RESULT_PASS ▪ TSS_LOG_RESULT_FAIL ▪ TSS_LOG_RESULT_WARN ▪ TSS_LOG_RESULT_STOPPED ▪ TSS_LOG_RESULT_INFO ▪ TSS_LOG_RESULT_COMPLETED ▪ TSS_LOG_RESULT_UNEVALUATED 0 specifies the default.
description	Specifies the string to be put in the entry's failure description field.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

An event and any data associated with it are logged only if the specified *`result`* preference matches associated settings in the `EVAR_LogData_control` (page 44) or `EVAR_LogEvent_control` (page 44) environment variables. Alternatively, the logging preference can be set with the `EVAR_Log_level` (page 45) and `EVAR_Record_level` (page 46) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

Example

This example logs the following message: `--Beginning of timed block T1--`.

```
TSSLog.message ("--Beginning of timed block T1--");
```

TSSLog.testCaseResult()

Logs a test case result.

Syntax

```
void testCaseResult (String testcase, short result, String
   description, TSSNamedValue[] property)

void testCaseResult (String testcase, short result)
```

Element	Description
testcase	Identifies the test case whose result is to be logged.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `TSS_LOG_RESULT_NONE` (default: no notification) ▪ `TSS_LOG_RESULT_PASS` ▪ `TSS_LOG_RESULT_FAIL` ▪ `TSS_LOG_RESULT_WARN` ▪ `TSS_LOG_RESULT_STOPPED` ▪ `TSS_LOG_RESULT_INFO` ▪ `TSS_LOG_RESULT_COMPLETED` ▪ `TSS_LOG_RESULT_UNEVALUATED` `0` specifies the default.
description	Contains the string to be displayed in the event of a log failure.
property	An array containing property name/value pairs, where `property[n].name` is the property name and `property[n].value` is its value. See *TSSNamedValue* on page 217 for the implementation of this argument's data type.

Exceptions

These methods may throw an exception with one of the following values:

- TSS_NOSERVER. No previous successful call to TSSSession.connect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

A test case is a condition, specified in a list of property name/value pairs, that you are interested in. This method searches for the test case and logs the result of the search.

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the EVAR_LogData_control (page 44) or EVAR_LogEvent_control (page 44) environment variables. Alternatively, the logging preference may be set by the EVAR_Log_level (page 45) and EVAR_Record_level (page 46) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

Example

This example logs the result of a testcase named Verify login.

```
TSSNamedValue[] loginResult = new TssNamedValue[1];
loginResult[0] = new TSSNamedValue();
loginResult[0].name = "Result";
loginResult[0].value = "OK";
TSSLog.testCaseResult("Verify login",0,null,loginResult);
```

Measurement Class

Use the measurement methods to set timers and environment variables, and to get the value of internal variables. Timers allow you to gauge how much time is required to complete specific activities under varying load conditions. Environment variables allow for the setting and passing of information to virtual testers during script playback. Internal variables store information used by the TestManager to initialize and reset virtual tester parameters during script playback.

Applicability

Commonly used with TestManager.

Summary

The following table lists the measurement methods. They are static methods of class `TSSMeasure`.

Method	**Description**
`commandEnd()`	Logs an end-command event.
`commandStart()`	Logs a start-command event.
`environmentOp()`	Sets an environment variable.
`getTime()`	Gets the elapsed time of a run.
`internalVarGet()`	Gets the value of an internal variable.
`think()`	Sets a think-time delay.
`timerStart()`	Marks the start of a block of actions to be timed.
`timerStop()`	Marks the end of a block of timed actions.

TSSMeasure.commandEnd()

Marks the end of a timed command.

Syntax

```
void commandEnd(short result, String description, int
   starttime, int endtime, String logdata, TSSNamedValue []
   property)

void commandEnd(short result)
```

Element	Description
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `TSS_LOG_RESULT_NONE` (default: no notification) ▪ `TSS_LOG_RESULT_PASS` ▪ `TSS_LOG_RESULT_FAIL` ▪ `TSS_LOG_RESULT_WARN` ▪ `TSS_LOG_RESULT_STOPPED` ▪ `TSS_LOG_RESULT_INFO` ▪ `TSS_LOG_RESULT_COMPLETED` ▪ `TSS_LOG_RESULT_UNEVALUATED`. `0` specifies the default.
description	Contains the string to be displayed in the event of failure.
starttime	An integer indicating a timestamp to override the timestamp set by `commandStart()`. To use the timestamp set by `commandStart()`, specify as `0`.
endtime	An integer indicating a timestamp to override the current time. To use the current time, specify as `0`.
logdata	Text to be logged describing the ended command.
property	An array containing property name/value pairs, where `property[n].name` is the property name and `property[n].value` is its value. See *TSSNamedValue* on page 217 for the implementation of this argument's data type.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The command name and label entered with `commandStart()` are logged, and the run state is restored to the value that existed before the `commandStart()` call.

An event and any data associated with it are logged only if the specified *`result`* preference matches associated settings in the `EVAR_LogData_control` (page 44) or `EVAR_LogEvent_control` (page 44) environment variables. Alternatively, the logging preference can be set with the `EVAR_Log_level` (page 45) and `EVAR_Record_level` (page 46) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

Example

This example marks the end of the timed activity specified by the previous `commandStart()` call.

```
TSSMeasure.commandEnd(TSS_LOG_RESULT_PASS,"Command timer failed", 0,
0, "Login command completed", null);
```

See Also

`commandStart(), TSSAdvanced.logCommand()`

TSSMeasure.commandStart()

Starts a timed command.

Syntax

```
void commandStart(String label, String name, int state)
```

Element	Description
label	The name of the timer to be started and logged, or NULL for an unlabeled timer.
name	The name of the command to time.

Element	Description
state	The run state to log with the timed command. See the run state table starting on page 78.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

A *command* is a term or string, such as `sock` or `deposit`, that you expect to occur in client/server conversations. By placing `commandStart()` and `commandEnd()` calls around expected strings, you can record the time required to complete associated actions.

During script playback, TestManager displays progress for different virtual testers. What is displayed for a group of actions associated by `commandStart()` depends on the run state argument. Run states are listed in the run state table starting on page 78.

`commandStart()` increments `IV_cmdcnt`, sets the name, label and run state for TestManager, and sets the beginning timestamp for the log entry. `commandEnd()` restores the TestManager run state to the run state that was in effect immediately before `commandStart()`.

Example

This example starts timing the period associated with the string `Login`.

```
TSSMeasure.commandStart("initTimer", "Login", MST_WAITRESP);
```

See Also

`commandEnd(), TSSAdvanced.logCommand()`

TSSMeasure.environmentOp()

Sets a virtual tester environment variable.

Syntax

void **environmentOpGetIntValue**(int *envVar*, int *envOp*, TSSInteger *envInt*)

void **environmentOpGetStringValue**(int *envVar*, int *envOp*, StringBuffer *envString*)

void **environmentOpSetIntValue**(int *envVar*, int *envOp*, int *envVal*)

Element	Description
envVar	The environment variable to operate on. Valid values are described in the environment variable table starting on page 43.
envOP	The operation to perform. Valid values are described in the environment operations table starting on page 50.
envInt	The new value for an integer environment variable. See *TSSInteger* on page 232 for the implementation of this argument's data type.
envString	The new value for a string environment variable.
envVal	The array index of the value to set the variable to.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The timer label is invalid, or there is no unlabeled timer to stop.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

Environment variables define and control the enviromnent of virtual testers. Using environment variables allows you to test different assumptions or runtime scenarios without re-writing your test scripts. For example, you can use environment variables to specify:

- A virtual tester's average think time, the maximum think time, and how the think time is mathematically distributed around a mean value
- How long to wait for a response from the server before timing out
- The level of information that is logged and available to reports

Use the `environmentOpGetIntValue()` for integer environment variables, `environmentOpGetStringValue()` for string environment variables, and `environmentOpSetIntValue()` for environment variables that can be set to one of a fixed number of values stored in an array.

The following table describes the valid values of argument *envVar*. Note the following about EVAR_LogData_control and EVAR_LogEvent_control:

- They correspond to the check boxes in TestManager's TSS Environment Variables dialog box. Use this dialog box to set logging and reporting options at the suite rather than the script level.
- They are more flexible alternatives to EVAR_Log_level and EVAR_Report_level.

Name	Type/Values/(default)	Contains
`EVAR_Delay_dly_scale`	integer 0–2000000000 percent (100)	The scaling factor applied globally to all timing delays. A value of 100%, which is the default, means no change. A value of 50% means one-half the delay, which is twice as fast as the original; 200% means twice the delay, which is half as fast. A value of zero means no delay.

Name	Type/Values/(default)	Contains

`EVAR_LogData_control`	`NONE,`
	`PASS,`
	`FAIL,`
	`WARNING,`
	`STOPPED,`
	`INFORMATIONAL,`
	`COMPLETED,`
	`UNEVALUATED`
	`ANYRESULT`
		Flags indicating the level of detail to log. Specify one or more. These result flags (except the last, which specifies everything) correspond to flags entered with the `event`, `message`, `testCaseResult`, `commandEnd`, and `logCommand` . For example, specifying `FAIL` selects everything logged by that specified flag FAIL.

`EVAR_LogEvent_control`	`NONE,`
	`PASS,`
	`FAIL,`
	`WARNING,`
	`STOPPED,`
	`INFORMATIONAL,`
	`COMPLETED,`
	`UNEVALUATED,`
	`TIMERS,`
	`COMMANDS,`
	`ENVIRON,`
	`STUBS,`
	`TSSERROR,`
	`TSSPROXYERROR`
	`ANYRESULT`
		Flags indicating the level of detail to log for reports. Specify one or more. The first nine result flags (NONE thru UNEVALUATED) correspond to flags specified with the `event`, `message`, `testCaseResult`, `commandEnd`, and `logCommand` . The other flags (TIMERS thru TSSPROXYERROR) indicate the event objects. For example, `FAIL` plus `COMMANDS` selects for reporting all commands that recorded a failed result. ANYRESULTS selects everything.

Name	Type/Values/(default)	Contains
`EVAR_Log_level`	string "OFF" ("TIMEOUT") "UNEXPECTED" "ERROR" "ALL"	The level of detail to log: ▪ `OFF` – Log nothing. ▪ `TIMEOUT` – Log emulation command timeouts. ▪ `UNEXPECTED` – Log timeouts and unexpected responses from emulation commands. ▪ `ERROR` – Log all emulation commands that set IV_`error` to a non-zero value. Log entries include IV_`error` and IV_`error_text`. ▪ `ALL` – Log everything: emulation command types and IDs, script IDs, source files, and line numbers.

Name	Type/Values/(default)	Contains
`EVAR_Record_level`	"MINIMAL" "TIMER" "FAILURE" ("COMMAND") "ALL"	The level of detail to log for reporting: ▪ `MINIMAL` – Record only items necessary for reports to run. Use this value when you do not want user activity to be reported. ▪ `TIMER` – `MINIMAL` plus `start_time` and `stop_time` emulation commands. Your reports will not contain response times for each emulation command, emulation command failure will not show up, and the result file for each virtual tester will be small. Use this setting if you are not concerned with the response times or pass/fail status of individual emulation commands. ▪ `FAILURE` – `TIMER` plus emulation command failures and some environment variable changes. Use this setting if you want the advantages of a small result file but you also want to make sure that no emulation command failed. ▪ `COMMAND` – `FAILURE` plus emulation command successes and some environment variable changes. ▪ `ALL` – `COMMAND` plus all environment variable changes. Complete recording.

Name	Type/Values/(default)	Contains
`EVAR_Suspend_check`	string ("ON") "OFF"	Controls whether you can suspend a virtual tester from a Monitor view: ▪ `ON` – A suspend request is checked before beginning the think time interval by each send emulation command. ▪ OFF – Disable suspend checking.
`EVAR_Think_avg`	integer 0–2000000000 ms (5000)	The average think-time delay (the amount of time that, on average, a user delays before performing an action).
`EVAR_Think_cpu_dly_scale`	integer 0–2000000000 ms (100)	The scaling factor applied globally to CPU (processing time) delays. Used instead of `EVAR_Think_dly_scale` if `EVAR_Think_avg` is less than `EVAR_Think_cpu_threshold`. Delay scaling is performed before truncation (if any) by `EVAR_Think_max`.
`EVAR_Think_cpu_threshold`	integer 0–2000000000 ms (0)	The threshold value used to distinguish CPU delays from think-time delays.

Name	Type/Values/(default)	Contains
EVAR_Think_def	string "FS" "LS" "FR" ("LR") "FC" "LC"	The starting point of the think-time interval: ▪ FS – the submission time of the previous send emulation command ▪ LS – the completion time of the previous send emulation command ▪ FR – the time the first data of the previous receive emulation command was received ▪ LR – the time the last data of the previous receive emulation command was received, or LS if there was no intervening receive emulation command ▪ FC – the submission time of the previous connect emulation command (uses the IV_fc_ts internal variable) ▪ LC – the completion time of the previous connect emulation command (uses the IV_lc_ts internal variable)

Name	Type/Values/(default)	Contains
`EVAR_Think_dist`	string (`"CONSTANT"`) `"UNIFORM"` `"NEGEXP"`	The think-time distrubution: ▪ `CONSTANT` – sets a constant distribution equal to `Think_avg` ▪ `UNIFORM` – sets a random think time interval distributed uniformly in the range: [`EVAR_Think_avg` - `EVAR_Think_sd`, `EVAR_Think_avg` + `EVAR_Think_sd`] ▪ `NEGEXP` – sets a random think time interval approximating a bell curve with `EVAR_Think_avg` equal to standard deviation
`EVAR_Think_dly_scale`	integer 0 – 2000000000 ms (100)	The scaling factor applied globally to think-time delays. Used instead of `EVAR_Think_cpu_dly_scale` if `EVAR_Think_avg` is greater than `EVAR_Think_cpu_threshold`. Delay scaling is performed before truncation (if any) by `EVAR_Think_max`.
`EVAR_Think_max`	integer 0–2000000000 ms (2000000000)	A maximum threshold for think times that replaces any larger setting.
`EVAR_Think_sd`	integer 0–2000000000 ms (0)	Where `EVAR_Think_dist` is set to UNIFORM, specifies the think time standard deviation.

Environment control options allow a script to control a virtual tester's environment by operating on the environment variables. Every environment variable has, instead of a single value, a group of values: a default value, a saved value, and a current value.

- **default** – The value of an environment variable before any commands are applied to it. Environment variables are automatically initialized to a default value, and, like persistent variables, retain their values across scripts. The `reset` command resets the default value, as listed in the following table.
- **saved** – The saved value of an environment variable can be used as one way to retain the present value of the environment variable for later use. The `save` and `restore` commands manipulate the saved value.
- **current** – TSS supports a last-in-first-out "value stack" for each environment variable. The current value of an environment variable is simply the top element of that stack. The current value is used by all of the commands. The `push` and `pop` commands manipulate the stack.

The following table describes the valid values of *`envOP`*.

Operation	**Description**
`EVOP_eval`	Operate on the value at the top of the variable's stack.
`EVOP_pop`	Remove the variable value at the top of the stack.
`EVOP_push`	Push a value to the top of a variable's stack.
`EVOP_reset`	Set the value of a variable to the default and discard any other values in the stack.
`EVOP_restore`	Set the saved value to the current value.
`EVOP_save`	Save the value of a variable.
`EVOP_set`	Set a variable to the specified value.

Example

This example turns off `EVAR_Suspend_check` before the start of a block of code and then turns it back on at the end of the block.

```
TSSMeasure.environmentOP (EVAR_Suspend_check, EVOP_push, "OFF");
//input emulation code //
TSSMeasure.evnironmentOP (EVAR_Suspend_check, EVOP_pop, "ON");
```

TSSMeasure.getTime()

Gets the elapsed time since the beginning of a suite run.

Syntax

```
int getTime()
```

Return Value

On success, this method returns the number of milliseconds elapsed in a suite run.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

For execution within TestManager, this call retrieves the time elapsed since the start time shared by all virtual testers in all test scripts in a suite.

For a test script executed outside TestManager, the time returned is the milliseconds elapsed since the call to `TSSSession.connect()`, or since the value of `CTXT_timeZero` set by `TSSSession.context()`.

Example

This example stores the elapsed time in *`etime`*.

```
int etime = TSSMeasure.getTime();
```

TSSMeasure.internalVarGet()

Gets the value of an internal variable.

Syntax

```
void internalVarGetInt(int internVar, TSSInteger iVal)
void internalVarGetString(int internVar, StringBuffer sVal)
```

Element	Description
internVar	The internal variable to operate on. Valid values are described in the string internal variables tableon page 52 and the integer internal variables table starting on page 53.
iVal	OUTPUT. The returned value of the specified integer internal variable. For the implementation of this argument's data type, see *TSSInteger* on page 232.
sVal	OUTPUT. The returned value of the specified string internal variable.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The timer label is invalid, or there is no unlabeled timer to stop.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

Internal variables contain detailed information that is logged during script playback and used for performance analysis reporting. This function allows you to customize logging and reporting detail.

The following table lists the string-valued internal variables that can be entered as argument *`sVal`*.

:

Variable	Contains
IV_alltext	The text of up to the value of Max_nrecv_saved. The same as IV_response.
IV_cmd_id	The ID of the most recent emulation command.
IV_column_headers	The two-line column header if IV_Column_headers is ON; otherwise, it contains "".
IV_command	The text of the most recent emulation command.
IV_error_text	The full text of the error from the last emulation command. If IV_error is 0, IV_error_text returns "". For an SQL database or TUXEDO error, the text is provided by the server.
IV_host	The host name of the computer on which the script is running.
IV_mcommand	The actual (mapped) sequence of characters submitted to the application by the most recent send or msend command. For send commands, IV_mcommand is always equivalent to IV_command.
IV_response	Same as IV_alltext.
IV_script	The name of the script currently being executed.
IV_source_file	The name of the file that was the source for the portion of the script being executed.
IV_user_group	The name of the user group (from the suite) of the user running the script.
IV_version	The full version string of TestManager (for example, 7.5.0.1045).

The following table lists the integer-valued internal variables that can be entered as argument *iVal*.

Variable	Contains
IV_cmdcnt	A running count of the number of emulation commands the script has executed.
IV_cursor_id	The last cursor declared by sqldeclare_cursor or opened by sqlopen_cursor.
IV_error	The status of the last emulation command. Most values for IV_error are supplied by the server.

Variable	Contains
`IV_error_type`	If you are emulating a TUXEDO session and `IV_error` is nonzero, `IV_error_type` contains one of the following values: 0 (no error) 1 VU/TUX Usage Error 2 TUXEDO System/T Error 3 TUXEDO FML Error 4 TUXEDO FML32 Error 5 application Error 6 Internal Error If you are emulating an IIOP session and `IV_error` is nonzero, `IV_error_type` contains one of the following values: 0 (no error) 1 IIOP_EXCEPTION_SYSTEM 2 IIOP_EXCEPTION_USER 3 IIOP_ERROR
`IV_fc_ts`	The "first connect" timestamp for `http_request` and `sock_connect`.
`IV_fr_ts`	The timestamp of the first received data of `sqlnrecv`, `http_nrecv`, `http_recv`, `http_header_recv`, `sock_nrecv`, or `sock_recv`. For `sqlexec` and `sqlprepare`, `IV_fr_ts` is set to the time the SQL database server responded to the SQL statement.
`IV_fs_ts`	The time the SQL statement was submitted to the server by `sqlexec` or `sqlprepare`, or the time when the first data was submitted to the server by `http_request` or `sock_send`.
`IV_lc_ts`	The "last connect" timestamp for `http_request` and `sock_connect`.
`IV_lineno`	The line number in `IV_source_file` of the previously executed emulation command.
`IV_lr_ts`	The timestamp of the last received data for `sqlnrecv`, `http_nrecv`, `http_recv`, `http_header_recv`, `sock_nrecv`, or `sock_recv`. For `sqlexec` and `sqlprepare`, `IV_lr_ts` is set to the time the SQL database server responded to the SQL statement.
`IV_ls_ts`	The time the SQL statement was submitted to the server by `sqlexec` or `sqlprepare`, or the time the last data was submitted to the server by `http_request` or `sock_send`.

Variable	Contains
`IV_nrecv`	The number of rows processed by the last `sqlnrecv`, or the number of bytes received by the last `http_nrecv`, `http_recv`, `sock_nrecv`, or `sock_recv`.
`IV_nusers`	The total number of virtual testers in the current TestManager session.
`IV_nxmit`	The total number of characters contained in the SQL statements transmitted to the server in the last `sqlexec` or `sqlprepare` command, or the number of bytes transmitted by the last `http_request` or `sock_send`.
`IV_statement_id`	The value assigned as the prepared statement ID, which is returned by `sqlprepare and sqlalloc_statement`.
`IV_total_nrecv`	The total number of bytes received for all HTTP and socket receive emulation commands issued on a particular connection.
`IV_total_rows`	Set to the number of rows processed by the SQL statements. If the SQL statements do not affect any rows, `IV_total_rows` is set to 0. If the SQL statements return row results, `IV_total_rows` is set to 0 by `sqlexec`, then incremented by `sqlnrecv` as the row results are retrieved.
`IV_tux_tpurcode`	TUXEDO user return code, which mirrors the TUXEDO API global variable `tpurcode`. It can be set only by the `tux_tpcall`, `tux_tpgetrply`, `tux_tprecv`, and `tux_tpsend` emulation commands.
`IV_uid`	The numeric ID of the current virtual tester.

Example

This example stores the current value of the `IVerror` internal variable in `IVVal`.

```
TSSMeasure.internalVarGet(IV_error,IVVal);
```

TSSMeasure.think()

Puts a time delay in a script that emulates a pause for thinking.

Syntax

```
void think(int thinkAverage)
void think()
```

Element	Description
thinkAverage	If specified as 0 or omitted, the number of milliseconds stored in the `EVAR_Think_avg` environment variable is used as the basis of the calculation. Otherwise, the calculation is based on the value specified.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

A think-time delay is a pause inserted in a performance test script in order to emulate the behavior of actual application users.

For a description of environment variables, see `environmentOp()` on page 42.

Example

This example calculates a pause based on the value stored in the environment variable `EVAR_Think_avg`, and inserts the pause into the script.

```
TSSMeasure.think();
```

See Also

`TSSAdvanced.thinkTime()`

TSSMeasure.timerStart()

Marks the start of a block of actions to be timed.

Syntax

```
void timerStart(String label, int timeStamp)
void timerStart(String label)
void timerStart()
```

Element	Description
label	The name of the timer to be inserted into the log. If specified as null, an unlabeled timer is created. Only one unlabeled timer is supported at a time.
timeStamp	An integer specifying a timestamp to override the current time. If specified as 0, the current time is logged.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This call associates a starting timestamp with *`label`* for later reference by `timerStop()`. The TestManager reporting system uses captured timing information for performance analysis reports.

Example

This example times actions designated `event1`, logging the current time.

```
TSSMeasure.timerStart ("event1");
// actions to be timed //
TSSMeasure.timerStop("event1");
```

See Also

`timerStop()`

TSSMeasure.timerStop()

Marks the end of a block of timed actions.

Syntax

```
void timerStop(String label, int timeStamp, boolean rmFlag)
void timerStop(String label)
void timerStop()
```

Element	Description
label	The name of the timer to be stopped and logged, or NULL for an unlabeled timer.
timeStamp	If specified as 0, the current time is recorded.
rmFlag	Specify as `false` (default) to stop the timer without removing it; otherwise, specify as `true`. A timer that is not removed can be stopped multiple times in order to measure intervals comprising this timed event.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The timer label is invalid, or there is no unlabeled timer to stop.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

Normally, this call associates an ending timestamp with a label specified with `timerStart()`. If the specified *`label`* was not set by a previous `timerStart()` but an unlabeled timer exists, this call uses the start time specified with `timerStart()` for the unlabeled timer. If `rmFlag` is specified as 0, multiple invocations of `timerStop()` are allowed against a single `timerStart()`. This usage (see the example) allows you to subdivide a timed event into separate timed intervals.

Example

This example stops an unlabeled timer without removing it.

```
TSSMeasure.timerStart();
// actions to be timed //
TSSMeasure.timerStop("event1");
// other actions to be timed //
TSSMeasure.timerStop("event2");
```

See Also

`timerStart()`

Utility Class

Use the utility methods to perform actions common to many test scripts.

Applicability

Commonly used with TestManager and QualityArchitect.

Summary

The following table lists the utility methods. They are static methods of class `TSSUtility`.

Method	**Description**
`delay()`	Delays the specified number of milliseconds.
`errorDetail()`	Retrieves error information about a failure.
`getScriptOption()`	Gets the value of a script playback option.
`getTestCaseConfigurationName()`	Gets the name of the configuration (if any) associated with the current test case.
`getTestCaseName()`	Gets the name of the test case in use.
`negExp()`	Gets the next negative exponentially distributed random number with the specified mean.
`rand()`	Gets the next random number.
`seedRand()`	Seeds the random number generator.
`stdErrPrint()`	Prints a message to the virtual tester′s error file.
`stdOutPrint()`	Prints a message to the virtual tester′s output file.
`uniform()`	Gets the next uniformly distributed random number in the specified range.

TSSUtility.delay()

Delays script execution for the specified number of milliseconds.

Syntax

```
void delay (int msecs)
```

Element	Description
msecs	The number of milliseconds to delay script execution.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The delay is scaled as indicated by the contents of the `EVAR_Delay_dly_scale` environment variable. The accuracy of the time delayed is subject to operating system limitations.

Example

This example delays execution for 10 milliseconds.

```
TSSUtility.delay(10);
```

TSSUtility.errorDetail()

Retrieves error information about a failure.

Syntax

```
int errorDetail(StringBuffer errorText)
```

Element	Description
errorText	OUTPUT. Returned explanatory error message about the previous TSS call, or an empty string ("") if the previous TSS call did not fail.

Return Value

This method returns `TSS_OK` if the previous call succeeded. If the previous call failed, TSSUtility.errorDetail() returns one of the error codes listed below and corresponding *`errorText`*.

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

This method is called internally by Java methods, which throw `TSSException` on error. Get the error code by calling `TSSException.getErrorCode()`. You can use `TSSUtility.errorDetail()`, but there is no need to do so. For implementation details, see *TSSException* on page 233.

Example

This example opens a datapool and, if there is an error, displays the associated error message text.

```
TSSDatapool dp = new TSSDatapool();
dp.open ("custdata");
StringBuffer errorText;
boolean fetchRet = dp.fetch();
if (fetchRet==false)
   // fetch failed, get detail
   int errorRet = TSSUtility.errorDetail (errorText);
   System.out.print (errorText);
```

TSSUtility.getScriptOption()

Gets the value of a script playback option.

Syntax

String **`getScriptOption`**(String *`optionName`*)

Element	Description
`optionName`	The name of the script option whose value is returned.

Return Value

On success, this method returns the value of the specified script option.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example gets the value of the script option `repeat_count`.

```
String optVal = TSSUtility.getScriptOption("repeat_count");
```

TSSUtility.getTestCaseConfigurationName()

Gets the name of the configuration (if any) associated with the current test case.

Syntax

```
String getTestCaseConfigurationName (void)
```

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

A test case specifies the pass criteria for something that needs to be tested. A configured test case is one that TestManager can execute and resolve as pass or fail.

Example

This example retrieves the name of a test case configuration.

```
String tcConfig = TSSUtility.getTestCaseConfigurationName();
```

TSSUtility.getTestCaseName()

Gets the name of the test case in use.

Syntax

```
String getTestCaseName()
```

Return Value

On success, this method returns the name of the current test case.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

Created from TestManager, a test case specifies the pass criteria for something that needs to be tested.

Example

This example stores the name of the test case in use in `tcName`.

```
String tcName = TSSUtility.getTestCaseName();
```

TSSUtility.negExp()

Gets the next negative exponentially distributed random number with the specified mean.

Syntax

```
int negExp(int mean)
```

Element	**Description**
mean	The mean value for the distribution.

Return Value

This method returns the next negative exponentially distributed random number with the specified mean.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.

- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

Example

This example seeds the generator and gets a random number with a mean of 10.

```
TSSUtility.seedRand (10)
int next = TSSUtility.negExp(10);
```

See Also

`rand(), seedRand(), uniform()`

TSSUtility.rand()

Gets the next random number.

Syntax

```
int rand()
```

Return Value

This method returns the next random number in the range 0 to 32767.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

Example

This example gets the next random number.

```
int next = TSSUtility.rand();
```

See Also

`seedRand(), negExp(), uniform()`

TSSUtility.seedRand()

Seeds the random number generator.

Syntax

```
void SeedRand(int seed)
```

Element	Description
seed	The base integer.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

`seedRand()` uses the argument *`seed`* as a seed for a new sequence of random numbers to be returned by subsequent calls to the `rand()` routine. If `seedRand()` is then called with the same seed value, the sequence of random numbers is repeated. If `rand()` is called before any calls are made to `seedRand()`, the same sequence is generated as when `seedRand()` is first called with a seed value of 1.

Example

This example seeds the random number generator with the number 10:

```
TSSUtility.seedRand(10);
```

See Also

`rand()`, `negExp()`, `uniform()`

TSSUtility.stdErrPrint()

Prints a message to the virtual tester's error file.

Syntax

```
void stdErrPrint(String message)
```

Element	Description
`message`	The string to print.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example prints to the error file the message `Login failed`.

```
TSSUtility.stdErrPrint("Login failed");
```

See Also

`TSSUtility.stdErrPrint()`

TSSUtility.stdOutPrint()

Prints a message to the virtual tester's output file.

Syntax

```
void stdOutPrint(String message)
```

Element	**Description**
message	The string to print.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example prints the message `Login successful`.

```
TSSUtility.stdOutPrint("Login successful");
```

See Also

`TSSUtility.stdErrPrint()`

TSSUtility.uniform()

Gets the next uniformly distributed random number.

Syntax

```
int uniform(int low, int high)
```

Element	Description
low	The low end of the range.
high	The high end of the range.

Return Value

This method returns the next uniformly distributed random number in the specified range, or –1 if there is an error.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The behavior of the random number generator routines is affected by the settings of the **Seed** and **Seed Flags** options in a TestManager suite. By default, TestManager sets unique seeds for each virtual tester, so that each has a different random number sequence.

Example

This example gets the next uniformly distriburted random number between –10 and 10.

```
int next = TSSUtility.uniform(-10,10);
```

See Also

`rand(), seedRand(), negExp()`

Monitor Class

When a suite of test cases or test scripts is played back, TestManager monitors execution progress and provides a number of monitoring options. The monitoring methods support TestManager's monitoring options.

Applicability

Commonly used with TestManager and QualityArchitect.

Summary

The following table lists the monitoring methods. They are static methods of class `TSSMonitor`.

Method	**Description**
`display()`	Sets a message to be displayed by the monitor.
`positionGet()`	Gets the script source file name or line number position.
`positionSet()`	Sets the script source file name or line number position.
`reportCommandStatus()`	Gets the runtime status of a command.
`runStateGet()`	Gets the run state.
`runStateSet()`	Sets the run state.

TSSMonitor.display()

Sets a message to be displayed by the monitor.

Syntax

`void` **`display`**`(String` *`message`*`)`

Element	**Description**
`message`	The message to be displayed by the progress monitor.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOOP`. The TSS server is running proxy.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This message will be displayed until overwritten by another call to `display()`.

Example

This example sets the monitor display to `Beginning transaction`.

```
TSSMonitor.display("Beginning transactioin");
```

TSSMonitor.positionGet()

Gets the test script file name or line number position.

Syntax

```
void positionGet (StringBuffer srcFile, TSSInteger lineNumber)
```

Element	Description
srcFile	OUTPUT. The name of a source file. After a successful call, this variable will contain the name of the source file that was specified with the most recent `positionSet()` call.
lineNumber	OUTPUT. The name of a local variable. After a successful call, this variable will contain the current line position in *srcFile* . For the implementation of this argument's data type, see *TSSInteger* on page 232.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

TestManager monitoring options include Script View, causing test script lines to be displayed as they are executed. `positionSet()` and `positionGet()` partially support this monitoring option for TSS scripts: if line numbers are reported, they will be displayed during playback but not the contents of the lines.

The line number returned by this function is the most recent value that was set by `positionSet()`. A return value of 0 for line number indicates that line numbers are not being maintained.

Example

This example gets the name of the current script file and the number of the line that will be accessed next.

```
StringBuffer scriptFile;
TSSInteger lineNumber;
TSSMonitor.positionGet(scriptFile,lineNumber);
```

See Also

`positionSet()`

TSSMonitor.positionSet()

Sets the test script file name or line number position.

Syntax

```
void positionSet(String srcFile, int lineNumber)
void positionSet(int lineNumber)
```

Element	Description
srcFile	The name of the test script, or NULL for the current test script.
lineNumber	The number of the line in *srcFile* to set the cursor to, or 0 for the current line.

Exceptions

These methods may throw an exception with one of the following values:

- TSS_NOSERVER. No previous successful call to TSSSession.connect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

TestManager monitoring options include Script View, causing test script lines to be displayed as they are executed. positionSet() and positionGet() partially support this monitoring option for TSS scripts: if line numbers are reported, they will be displayed during playback but not the contents of the lines.

Example

This example sets access to the beginning of test script checkLogin.

```
TSSMonitor.positionSet("checkLogin",0);
```

See Also

positionSet()

TSSMonitor.reportCommandStatus()

Reports the runtime status of a command.

Syntax

```
void reportCommandStatus(int status)
```

Element	Description
status	The status of a command. Can be one of the following: ▪ `TSS_CMD_STAT_FAIL` ▪ `TSS_CMD_STAT_PASS` ▪ `TSS_CMD_STAT_WARN` ▪ `TSS_CMD_STAT_INFO`.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOOP`. The TSS server is running proxy.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The entered *status* is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Example

This example reports a failure command status.

```
TSSMonitor.reportCommandStatus(TSS_CMD_STAT_FAIL);
```

TSSMonitor.runStateGet()

Gets the run state.

Syntax

```
int runStateGet()
```

Return Value

On success, this method returns one of the run state values listed in the run state table starting on page 78.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This call is useful for storing the current run state so you can change the state and then subsequently do a reset to the original run state.

Example

This example gets the current run state.

```
int orig = TSSMonitor.runStateGet();
```

See Also

`runStateSet()`

TSSMonitor.runStateSet()

Sets the run state.

Syntax

```
void runStateSet(int state)
```

Element	Description
state	The run state to set. Enter one of the run state values listed in the run state table starting on page 78.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. Invalid run state.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

TestManager includes the option to monitor script progress individually for different virtual testers. The run states are the mechanism used by test scripts to communicate their progress to TestManager. Run states can also be logged and can contribute to performance analysis reports.

The following table lists the TestManager run states.

Run State	Meaning
`MST_BIND`	iiop_bind in progress
`MST_BUTTON`	X button action
`MST_CLEANUP`	cleaning up
`MST_CPUDLY`	cpu delay
`MST_DELAY`	user requested delay
`MST_DSPLYRESP`	displaying response
`MST_EXITED`	exited

Run State	Meaning
MST_EXITSQABASIC	exited SQABasic code
MST_EXTERN_C	executing external C code
MST_FIND	find_text find_point
MST_GETTASK	waiting for task assignment
MST_HTTPCONN	waiting on http connection
MST_HTTPDISC	waiting on http disconnect
MST_IIOP_INVOKE	iiop_invoke in progress
MST_INCL	mask including above basic states
MST_INITTASK	initializing task
MST_ITDLY	inter-task delay
MST_MOTION	X motion
MST_PMATCH	matching response (precv)
MST_RECV_DELAY	line_speed delay in recv
MST_SATEXEC	executing satellite script
MST_SEND	httpsocket send
MST_SEND_DELAY	line_speed delay in send
MST_SHVBLCK	blocked from shv access
MST_SHVREAD	V_VP: reading shared variable
MST_SHVWAIT	user requested shv wait
MST_SOCKCONN	waiting on socket connection
MST_SOCKDISC	waiting on socket disconnect
MST_SQABASIC_CODE	running SQABasic code
MST_SQLCONN	waiting on SQL client connection
MST_SQLDISC	waiting on SQL client disconnect
MST_SQLEXEC	executing SQL statements
MST_STARTAPP	SQABasic: starting app
MST_SUSPENDED	suspended

Run State	Meaning
`MST_EXITSQABASIC`	exited SQABasic code
`MST_EXTERN_C`	executing external C code
`MST_FIND`	find_text find_point
`MST_GETTASK`	waiting for task assignment
`MST_HTTPCONN`	waiting on http connection
`MST_HTTPDISC`	waiting on http disconnect
`MST_IIOP_INVOKE`	iiop_invoke in progress
`MST_INCL`	mask including above basic states
`MST_INITTASK`	initializing task
`MST_ITDLY`	inter-task delay
`MST_MOTION`	X motion
`MST_PMATCH`	matching response (precv)
`MST_RECV_DELAY`	line_speed delay in recv
`MST_SATEXEC`	executing satellite script
`MST_SEND`	httpsocket send
`MST_SEND_DELAY`	line_speed delay in send
`MST_SHVBLCK`	blocked from shv access
`MST_SHVREAD`	V_VP: reading shared variable
`MST_SHVWAIT`	user requested shv wait
`MST_SOCKCONN`	waiting on socket connection
`MST_SOCKDISC`	waiting on socket disconnect
`MST_SQABASIC_CODE`	running SQABasic code
`MST_SQLCONN`	waiting on SQL client connection
`MST_SQLDISC`	waiting on SQL client disconnect
`MST_SQLEXEC`	executing SQL statements
`MST_STARTAPP`	SQABasic: starting app
`MST_SUSPENDED`	suspended

Run State	Meaning
MST_TEST	test case, emulate
MST_THINK	thinking
MST_TRN_PACING	transactor pacing delay
MST_TUXEDO	Tuxedo execution
MST_TYPE	typing
MST_USERCODE	SQAVu user code
MST_INIT	doing start-up initialization
MST_UNDEF	user's micro_state is undefined
MST_WAITOBJ	SQABasic: waiting for object
MST_WAITRESP	waiting for response
MST_WATCH	interactive -W watch record
MST_XCLNTCONN	waiting on http connection
MST_XCLNTCONN	waiting on socket connection
MST_XCLNTCONN	waiting on SQL client connection
MST_XCLNTCONN	waiting on X client connection
MST_XCLNTDISC	waiting on http disconnect
MST_XCLNTDISC	waiting on socket disconnect
MST_XCLNTDISC	waiting on SQL client disconnect
MST_XCLNTDISC	waiting on X client disconnect
MST_XMOVEWIN	X move window
MST_XQUERY	X query function
MST_XSYNC	X sync state during X query
MST_XWINCMP	xwindow_diff comparing windows
MST_XWINDUMP	xwindow_diff dumping window
N_MST_INCL	number of above states

Example

This example sets the run state to `MST_WAITRESP`.

```
TSSMonitor.runStateSet(MST_WAITRESP);
```

See Also

`runStateGet()`

Synchronization Class

Use the synchronization methods to sychronize virtual testers during script playback. You can insert synchronization points and wait periods, and you can manage variables shared among virtual testers.

Applicability

Commonly used with TestManager.

Summary

The following table lists the synchronization methods. They are static methods of class `TSSSync`.

Method	**Description**
`sharedVarAssign()`	Performs a shared variable assignment operation.
`sharedVarEval()`	Gets the value of a shared variable and operates on the value as specified.
`sharedVarWait()`	Waits for the value of a shared variable to match a specified range.
`syncPoint()`	Puts a synchronization point in a script.

TSSSync.sharedVarAssign()

Performs a shared variable assignment operation.

Syntax

```
void sharedVarAssign(String name, int val, int op, TSSInteger
   returnVal)
```

Element	Description
name	The name of the shared variable to operate on.
value	The right-hand-side value of the assignment expression.
op	Assignment operator. Can be one of the following: ▪ `SHVOP_assign` ▪ `SHVOP_add` ▪ `SHVOP_subtract` ▪ `SHVOP_multliply` ▪ `SHVOP_divide` ▪ `SHVOP_modulo` ▪ `SHVOP_and` ▪ `SHVOP_or` ▪ `SHVOP_xor` ▪ `SHVOP_shiftleft` ▪ `SHVOP_shiftright`
returnVal	OUTPUT. If not specified as NULL, the resulting value of *name* after application of *op value*.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The entered *name* is not a shared variable.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

Shared variables require configuration. For details, see the following example and Appendix A.

Example

This example adds 5 to the value of the shared variable `lineCounter`, puts the new value of `lineCounter` in `returnval`, and configures the variable by adding it to an array naming all shared variables used in the script. This configuration code can occur anywhere in the script.

```
TSSInteger returnVal = new TSSIngeger(0);
TSSSync.sharedVarAssign("lineCounter", 5, SHVOP_add, returnVal);
...
public static class SharedVarConfig extends SharedVarInfo {
   public SharedVarConfig() {
      String sv[] = {
         "lineCounter",
         ....
      }
      setSharedVarNames(sv);
   }
}
```

See Also

`sharedVarEval(), sharedVarWait()`

TSSSync.sharedVarEval()

Gets the value of a shared variable and operates on the value as specified.

Syntax

```
void sharedVarEval(String name, TSSInteger value, int op)
```

Element	Description
name	The name of the shared variable to operate on.
value	OUTPUT. A local container into which the value of `name` is retrieved. For the implementation of this argument's data type, see *TSSInteger* on page 232.

Element	Description
op	Increment/decrement operator for the returned value: Can be one of the following: ▪ `SHVADJ_none` `SHVADJ_pre_inc` ▪ `SHVADJ_post_inc` ▪ `SHVADJ_pre_dec` ▪ `SHVADJ_post_dec`

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The entered *`name`* is not a shared variable.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

Shared variables require configuration. For details, see Appendix A.

Example

This example post-decrements the value of shared variable `lineCounter` and stores the result in `val`.

```
TSSInteger val = new TSSInteger(0);
TSSSync.sharedVarEval("lineCounter",val,SHVADJ_post_inc);
```

See Also

`sharedVarAssign(), sharedVarWait()`

TSSSync.sharedVarWait()

Waits for the value of a shared variable to match a specified range.

Syntax

```
int sharedVarWait(String name, int min, int max, int adjust,
   int timeout, TSSInteger returnVal)
```

```
int sharedVarWait(String name, int min)
```

Element	Description
name	The name of the shared variable to operate on.
min	The low range for the value of *name*.
max	The high range for the value of *name*.
adjust	The value to increment/decrement the named shared variable by once it meets the *min* – *max* range.
timeout	The timeout preference (how long to wait for the condition to be met). Enter one of the following: ▪ A negative number for no timeout. ▪ 0 to return immediately with an exit value of `1` (condition met) or `0` (not met) ▪ The number of milliseconds to wait for the value of *name* to meet the criteria, before timing out with and returning an exit value of `1` (met) or `0` (not met).
returnVal	OUTPUT. The value of *name* at the time of the return, before any possible adjustment. If *timeout* expired before the return, the value is not adjusted. Otherwise, *returnVal* is incremented/decremented by *adjust*. For the implementation of this argument's data type, see *TSSInteger* on page 232.

Return Value

On success, this method returns `1` (condition was met before timeout) or `0` (timeout expired before the condition was met).

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The entered *name* is not a shared variable.

- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

This call provides a method of blocking a virtual tester until a user-defined global event occurs.

If virtual testers are blocked on an event utilizing the same shared variable, TestManager guarantees that the virtual testers are unblocked in the same order in which they were blocked. Although this *alone* does not ensure an exact multi-user timing order in which statements following a `wait` are executed, the additional proper use of the arguments *`min`*, *`max`*, and *`adjust`* allows control over the order in which multi-user operations occur. (UNIX or Windows NT determines the order of the scheduling algorithms. For example, if two virtual testers are unblocked from a wait in a given order, the tester that was unblocked last might be released before the tester that was unblocked first.)

If a shared variable's value is modified, any subsequent attempt to modify this value — other than through `sharedVarWait()` — blocks execution until all virtual testers already blocked have had an *opportunity* to unblock. This ensures that events cannot appear and then quickly disappear before a blocked virtual tester is unblocked. For example, if two virtual testers were blocked waiting for *`name`* to equal or exceed *`N`*, and if another virtual tester assigned the value *`N`* to *`name`*, then TestManager guarantees both virtual testers the opportunity to unblock before any other virtual tester is allowed to modify *`name`*.

Offering the *opportunity* for all virtual testers to unblock does not guarantee that all virtual testers actually unblock, because if `sharedVarWait()` is called with a nonzero value of *`adjust`* by one or more of the blocked virtual testers, the shared variable value changes during the unblocking script. In the previous example, if the first user to unblock *had* called `sharedVarWait()` with a negative *`adjust`* value, then the event waited on by the second user would no longer be true after the first user unblocked. With proper choice of *`adjust`* values, you can control the order of events.

Shared variables require configuration. For details, see Appendix A.

Example

This example returns `1` if the shared variable `inProgress` reaches a value between 10 and 20 within 60000 milliseconds of the time of the call. Otherwise, it returns `0`. `svVal` contains the value of `inProgress` at the time of the return, before it is adjusted. (In this case, the adjustment value is 0 so the value of the shared variable is not adjusted.)

```
TSSInteger svVal = new TSSInteger(0);
int retVal= TSSSync.sharedVarWait("inProgress",10,20,0,60000,svVal);
```

See Also

`sharedVarAssign(), sharedVarEval()`

TSSSync.syncPoint()

Puts a synchronization point in a script.

Syntax

`void` **`syncPoint`**`(String` *`label`*`)`

Element	Description
label	The name of the synchronization point.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOOP`. The TSS server is running proxy.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The synchronication point *`label`* is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

A script pauses at a synchronization point until the release criteria specified by the suite have been met. If the criteria are met, the script delays a random time specified in the suite and then resumes execution.

Typically, you will want to insert synchronization points into a TestManager suite rather than inserting the `syncPoint()` call into a script.

If you insert a synchronization point into a suite, synchronization occurs at the beginning of the script. If you insert a synchronization point into a script with `syncPoint()`, synchronization occurs at the point of insertion. You can insert the command anywhere in the script.

Shared variables require configuration. For details, see the following example and Appendix A.

Example

This example creates a sync point named `BlockUntilSaveComplete`and configures the sync point. The configuration statement may appear anywhere inside the script.

```
TSSSync.syncPoint("BlockUntilSaveComplete");
...
public static class SyncPointConfig extends SyncPointInfo {
        public SyncPointConfig() {
            String points[] = {
                "BlockUntilSaveComplete"};
            setSyncPointNames(points);
        }
}
```

Session Class

A suite can contain multiple test scripts of different types. When TestManager executes a suite, a separate *session* is started for each type of script in the suite. Each session lasts until all scripts of the type have finished executing. Thus, if a suite contains three Visual Basic test scripts and six VU test scripts, two sessions will be started and each will remain active until all scripts of the respective types finish.

In a given suite run, a session can be run directly (inside TestManager's process space) or by a separate TSS server process (proxy). The latter will happen only if the following two conditions are met:

- The test script(s) is executed stand-alone (outside of TestManager) and is linked with the link library rttssremote.lib.
- The first script of a given type in a suite that can be executed by a TSS proxy server calls `serverStart()`.

Unlike most TSS methods, the Session methods do not generate error codes or throw exceptions. Instead, they return status values indicating success or the cause of failure.

Applicability

Commonly used with TestManager.

Summary

Use the session methods listed in the following table to manage proxy TSS servers and sessions. These methods are not needed for sessions that are directly executed by TestManager. These are static methods of class `TSSSession`.

Method	**Description**
`connect()`	Connects to a TSS proxy server.
`context()`	Passes context information to a TSS server.
`disconnect()`	Disconnects from a TSS proxy server.
`serverStart()`	Starts a TSS proxy server.
`serverStop()`	Stops a TSS proxy server.
`shutdown()`	Stops logging and initializes TSS.

TSSSession.connect()

Connects to a TSS proxy server.

Syntax

```
int connect(String host, int port, int id)
```

Element	Description
host	The name (or IP address in quad dot notation) of the host on which the proxy TSS server process is running.
port	The listening port for the TSS server on *host*, or 0 (recommended) to let TestManager select the port.
id	The connection identifier.

Return Value

- TSS_OK. Success.
- TSS_NOOP. A connection and ID had already been established for this execution thread.

Exceptions

This method may throw an exception with one of the following values:

- TSS_NOSERVER. No TSS server was listening on *port*.
- TSS_SYSERROR. A system error occurred.
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

Comments

For scripts that are executed by a proxy process rather than directly by the TSEE, this function must be called before any other TSS functions. This function is also required when a script starts a new thread of execution.

The direct TSS DLL ignores *host* and *port*, and associates the *id* with the current execution thread. If the thread already had an ID, then *id* is ignored. (You cannot change *id*.)

Example

This example connects to a TSS server running on host 192.36.25.107. The *port* is defined in the example for `serverStart()`.

```
TSSInteger port = new TSSInteger(0);
int retval = TSSSession.connect ("192.36.25.107",port.getValue(),0);
```

See Also

`serverStart()`

TSSSession.context()

Passes context information to a TSS server.

Syntax

int **`context`** (int *`ctx`*, String *`value`*)

Element	**Description**
ctx	The type of context information to pass: Can be one of the following: ▪ `CTXT_workingDir` ▪ `CTXT_datapoolDir` ▪ `CTXT_timeZero` ▪ `CTXT_todZero` ▪ `CTXT_logDir` ▪ `CTXT_logFile` ▪ `CTXT_logData` ▪ `CTXT_testScript` ▪ `CTXT_style` ▪ `CTXT_sourceUID`
value	The information of type *`ctx`* to pass.

Return Value

- `TSS_OK`. Success.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The specified *`ctx`* is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

This call is useful for test scripts that are executed stand-alone — outside the TestManager framework — and that also make TSS calls. The call passes information, such as the log file name, that would be passed through shared memory if the script were executed by TestManager.

Test scripts that are executed stand-alone and also by a proxy TSS server should make this call immediately after `TSSSession.connect()`, before accessing any other TSS services. Otherwise, inconsistent results can occur.

Example

This example passes a working directory to the current proxy TSS server.

```
int retVal = TSSSession.context(CTXT_workingDir,"C:\temp");
```

TSSSession.disconnect()

Disconnects from a TSS proxy server.

Syntax

```
void disconnect()
```

Return Value

None.

Comments

This call closes the connection established by `TSSSession.Cconnect()` and performs any required cleanup operations.

Example

This example disconnects from the TSS server.

```
TSSSession.disconnect ();
```

TSSSession.serverStart()

Starts a TSS proxy server.

Syntax

```
int serverStart(TSSInteger port)
```

Element	Description
port	The listening port for the TSS server. If specified as `0` (recommended), the system chooses the port and returns its number to *port*. See *TSSInteger* on page 232 for the implementation of this argument's type.

Return Value

This method does not throw an exception on error. A script may check for one of the following return values.

- `TSS_OK`. Success.
- `TSS_NOOP`. A TSS server was already listening on *port*.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_NOSERVER`. Start failure.
- `TSS_SYSERROR`. A system error occurred.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

No TSS server is started if one is already running. A test script that is to be executed by a proxy server and that might be the first to execute, should make this call.

Example

This example starts a proxy TSS server on a system-designated port, whose number is returned to *port*.

```
TSSInteger port = new TSSInteger(0);
int retVal = TSSSession.serverStart (port);
```

See Also

`serverStop()`

TSSSession.serverStop()

Stops a TSS proxy server.

Syntax

```
int serverStop(int port)
```

Element	Description
port	The port number that the TSS server to be stopped is listening on.

Return Value

- `TSS_OK`. Success.
- `TSS_NOOP`. No TSS server was listening on *port*.

Exceptions

This method may throw an exception with one of the following values:

- `TSS_INVALID`. No proxy TSS server was found or stopped.
- `TSS_SYSERROR`. A system error occurred.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

In a test suite with multiple scripts, only the last executed script should make this call.

Example

This example stops a proxy TSS server that was started by the example for `serverStart()`.

```
int retVal = TSSSession.serverStop (port.getValue());
```

See Also

`serverStart()`

TSSSession.shutdown()

Stops logging and initializes TSS.

Syntax

```
void shutdown(void)
```

Return Value

This method exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The specified *`ctx`* is invalid.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

Comments

This call stops logging functions, pauses a playback session, and initializes TSS to resume logging and executing the next task.

Example

This example shuts down logging during session execution so that logging can be restarted for the next task.

```
int retVal = TSSSession.shutdown ();
```

Advanced Class

You can use the advanced methods to perform timing calculations, logging operations, and internal variable initialization functions. TestManager performs these operations on behalf of scripts in a safe and efficient manner. As a result, the functions need not and usually should not be performed by individual test scripts.

Applicability

Commonly used with TestManager.

Summary

The following table lists the advanced methods. They are static methods of class `TSSAdvanced`.

Method	**Description**
`internalVarSet()`	Sets the value of an internal variable.
`logCommand()`	Logs a command event.
`thinkTime()`	Calculates a think-time average.

TSSAdvanced.internalVarSet()

Sets the value of an internal variable.

Syntax

```
void internalVarSetInt(int internVar, int iVal)
void internalVarSetString(int internVar, StringBuffer sVal)
```

Element	Description
internVar	The internal variable to operate on. Internal variables and their values are listed in the tables starting on page 52 and page 53.
iVal	The new integer value for *internVar*. For the implementation of this argument's data type, see *TSSInteger* on page 232.
sVal	The new string internal value for *internVar*.

Exceptions

These methods may throw an exception with one of the following values:

- `TSS_NOSERVER`. No previous successful call to `TSSSession.connect()`.
- `TSS_INVALID`. The timer label is invalid, or there is no unlabeled timer to stop.
- `TSS_ABORT`. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The values of some internal variables affect think-time calculations and the contents of log events. Setting a value incorrectly could cause serious misbehavior in a script.

Example

This example sets `IV_cmdcnt` to 0.

```
TSSAdvanced.internalVarSetInt (IV_cmdcnt,0);
```

See Also

`TSSMeasure.internalVarGet()`

TSSAdvanced.logCommand()

Logs a command event.

Syntax

```
void logCommand(String name, String label, short result, String
   description, int starttime, int endtime, String logdata,
   TSSNamedValue [] property)
```

```
void logCommand(String name, String label, short result)
```

Element	Description
name	The command name.
label	The event label.
result	Specifies the notification preference regarding the result of the call. Can be one of the following: ▪ `TSS_LOG_RESULT_NONE` (default: no notification) ▪ `TSS_LOG_RESULT_PASS` ▪ `TSS_LOG_RESULT_FAIL` ▪ `TSS_LOG_RESULT_WARN` ▪ `TSS_LOG_RESULT_STOPPED` ▪ `TSS_LOG_RESULT_INFO` ▪ `TSS_LOG_RESULT_COMPLETED` ▪ `TSS_LOG_RESULT_UNEVALUATED` `0` specifies the default.
description	Contains the string to be displayed in the event of failure.
starttime	An integer indicating a timestamp. If specified as 0, the logged timestamp will be the later of the values contained in internal variables `IV_fcs_ts` and `IV_fcr_ts`.
endtime	An integer indicating a timestamp. If specified as 0, the time set by `commandEnd()` is logged.
logdata	Text to be logged describing the ended command.
property	An array containing property name/value pairs, where `property[n].name` is the property name and `property[n].value` is its value. For the implementation of this argument's data type, see *TSSNamedValue* on page 217.

Exceptions

These methods may throw an exception with one of the following values:

- TSS_NOSERVER. No previous successful call to TSSSession.connect().
- TSS_ABORT. Pending abort resulting from a user request to stop a suite run.

If you handle one of these exceptions and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass result for the script.

Comments

The value of IV_cmdcnt is logged with the event.

The command name and label entered with TSSMeasure.commandStart() are logged, and the run state is restored to the value that existed prior to the TSSMeasure.commandStart() call.

An event and any data associated with it are logged only if the specified *result* preference matches associated settings in the EVAR_LogData_control (page 44) or EVAR_LogEvent_control (page 44) environment variables. Alternatively, the logging preference may be set with the EVAR_Log_level (page 45) and EVAR_Record_level (page 46) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

Example

This example logs a message for a login script.

```
TSSAdvanced.logCommand("Login", "initTimer", TSS_LOG_RESULT_PASS,
"Command timer failed", 0, 0, "Login command completed", null);
```

See Also

TSSMeasure.commandStart(),TSSMeasure.commandEnd()

TSSAdvanced.thinkTime()

Calculates a think-time average.

Syntax

```
int thinkTime(int thinkAverage)
int thinkTime()
```

Element	Description
thinkAverage	If specified as 0, the number of milliseconds stored in the `ThinkAvg` environment variable is entered. Otherwise, the value specified overrides `ThinkAvg`.

Return Value

On success, these methods return a calculated think-time average.

Comments

This call calculates and returns a think time using the same algorithm as `TSSMeasure.think()`. But unlike `TSSMeasure.think()`, this call inserts no pause into a script.

This function could be useful in a situation where a test script calls another program that, as a matter of policy, does not allow a calling program to set a delay in execution. In this case, the called program would use `TSSMeasure.thinkTime()` to recalculate the delay requested by `TSSMeasure.think()` before deciding whether to honor the request.

Example

This example calculates a pause based on a think-time average of 5000 milliseconds.

```
ctime = ‘tsscmd GetTime‘
int iv = TSSMeasure.getTime();
TSSAdvanced.internalVarSetInt(IV_fcs_ts, iv);
TSSAdvanced.internalVarSetInt(IV_lcs_ts, iv);
TSSAdvanced.internalVarSetInt(IV_fcr_ts, iv);
TSSAdvanced.internalVarSetInt(IV_lcr_ts, iv);
int pause = TSSAdvanced.thinkTime(5000);
```

See Also

`TSSMeasure.think()`

Extended Test Script Services Reference 4

About the Extensions

This chapter describes two classes that extend some of the functionality of the Rational Test Script Services (TSS):

- *LookUpTable Class* on page 104

  The `LookUpTable` class is designed for use with Rational QualityArchitect stubs.

- *TestLog Class* on page 112

  This class extends `TSSLog`. It is designed to let you log information from Rational QualityArchitect test scripts and stubs. However, you can use this class to log information from any program.

Requirements for Using the Test Script Services Extensions

The Test Script Services extensions described in this chapter require Rational Quality Architect.

In addition, the CLASSPATH must reference a number of .jar files. For a list of the required .jar files, see *Running Test Scripts* on page 8.

LookUpTable Class

This class lets a method in a stub access a lookup table.

A *lookup table* lets you test a component whose operation depends upon an associated component that is still in the development stages. To test the component, you first provide a stub of the unfinished component that contains that component's methods. When the component-under-test calls a method in the stub, the method simulates operation by retrieving information from the lookup table — information that would otherwise be generated during normal execution in the completed component. The method then presents the retrieved information to the calling component-under-test.

The information that a stub's method retrieves from the lookup table depends upon the values that the component-under-test passes to the method. In other words, a method finds the lookup-table row that contains the parameter values that the component-under-test passed to it, and then retrieves the appropriate value (return value or exception) from that same lookup-table row.

A lookup table typically has multiple rows, with each row representing a different set of inputs and outputs. This allows a method in the component-under-test to be executed multiple times against the stub, supplying different input values and retrieving different output values each time.

In the following example of a lookup table for a mortgage calculation method, `amount`, `interest`, and `months` are input values, while `expectedReturn` and `expectedException` are the corresponding output values:

amount	interest	months	expectedReturn	expectedException
100000	0.0700	240	775.30	
125000	00725	300		bank.BadRateException
150000	0.0750	360	1048.83	

Typically, you create a lookup table for each stub method that is called during testing of the component-under-test.

The underlying files used for both lookup tables and datapools are the same. As a result, when it is time to replace the stub with the completed component, you can use the lookup table as a datapool when you test the associated component-under-test.

Note: A stub is not a test script. Consequently, it does not require a `testMain()` method.

Overview

```
public class LookUpTable extends java.lang.Object

java.lang.Object
  |
  +--com.rational.test.ct.LookUpTable
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

LookUpTable Example

The following sample code opens and retrieves values from the lookup table `ManageAccountsBean_getSavingsBalance_L`. The code contains examples of both `LookUpTable` methods and `TestLog` methods. `TestLog` methods are described in *TestLog Class* on page 112.

```
public java.math.BigDecimal getSavingsBalance(long accountID) throws
      java.rmi.RemoteException,javax.naming.NamingException,
      javax.ejb.EJBException {
   String[] ParamNames = new String[1];
   ParamNames[0] = "accountID";
   return getSavingsBalance_lookup(ParamNames, accountID);
}

private java.math.BigDecimal getSavingsBalance_lookup(
      String[] ParamNames,long accountID) throws
      java.rmi.RemoteException,javax.naming.NamingException,
      javax.ejb.EJBException{
   java.math.BigDecimal retval = null;
   TestLog log = new TestLog();
   LookUpTable lookup = new LookUpTable();
   String sRetval = null;
   try
   {
      lookup.open("ManageAccountsBean_getSavingsBalance_L");
      String[] values = new String[1];
      values[0] = Long.toString(accountID);
      log.writeStubMessage(
            "ManageAccounts stub, getSavingsBalance method. ",
            "Entered with following values: " + values[0] + " " + " ");
      if (lookup.find(ParamNames, values))
      {
         Exception eExpected = lookup.getExpectedException();
         if (eExpected != null)
         {
            log.writeStubMessage(
                  "ManageAccounts stub, getSavingsBalance method. ",
                 "Throwing exception: " +
                 eExpected.getClass().getName());
             throw eExpected;
           }
           else
           {
             sRetval = lookup.getReturnValue();
             if (sRetval != null)
               retval = new java.math.BigDecimal(sRetval);
;
           }
         }
         else
         {
           Exception ex = new Exception("Entry could not be found in the
               lookup table for  ManageAccounts stub, getSavingsBalance
               method.");
           log.writeStubException("Lookup table entry not found error: ",
               ex);
         }
         lookup.close();
       }
       catch (java.rmi.RemoteException e)
       {
         throw e;
       }
       catch (javax.naming.NamingException e)
       {
         throw e;
       }
       catch (Exception e)
       {
         log.writeStubException("Lookup table Error in ManageAccounts
             stub,getSavingsBalance method: ", e);
       }
       log.writeStubMessage("ManageAccounts stub, getSavingsBalance
             method. ", "Returning " + sRetval);
       return retval;
     }
```

Summary

This class contains the following methods:

Method	**Description**
`close()`	Closes the current lookup table (that is, the lookup table associated with this instance of the `LookUpTable` class).
`find()`	Sets the cursor to the row in the current lookup table that contains the column value(s) passed to it.
`getExpectedException()`	Returns the contents of the expectedException column in the current lookup table row.
`getReturnValue()`	Returns the contents of the expectedReturn column in the current lookup table row.
`getValue()`	Returns the contents of the specified column in the current lookup table row.
`open()`	Opens the specified lookup table.

Constructor

Syntax

```
public LookUpTable()
```

LookUpTable.close()

Closes the current lookup table (that is, the lookup table associated with this instance of the LookUpTable class).

Syntax

```
public void close()
```

Example

For an example of this method, see *LookUpTable Example* on page 105.

LookUpTable.find()

Sets the cursor to the row in the current lookup table that contains the column value(s) passed to it.

Syntax

```
public boolean find(java.lang.String[] names,
   java.lang.String[] values)
```

Element	Description
names	An array containing one or more lookup-table column names.
values	An array containing a value for each corresponding column name passed to the method.

Return Value

If `true`, the cursor was successfully set to the row that matched the specified criteria. If `false`, a row could not be found that matched the specified criteria.

Exceptions

This method throws the following exception:

- `java.lang.Exception`. Reports problems attempting to set the cursor to a row in the lookup table.

If you handle this exception and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass Result for the script.

Comments

Subsequent value-retrieval methods act upon the row with the cursor.

If multiple rows contain the passed value(s), this method throws an exception.

Example

For an example of this method, see *LookUpTable Example* on page 105.

LookUpTable.getExpectedException()

Returns the contents of the expectedException column in the current lookup table row.

Syntax

```
public java.lang.Exception getExpectedException()
```

Return Value

The contents of the Exception column in the current lookup table row.

Exceptions

This method throws the following exception:

- `java.lang.Exception`. Reports problems attempting to retrieve the contents of the Exception column.

If you handle this exception and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass Result for the script.

Example

For an example of this method, see *LookUpTable Example* on page 105.

LookUpTable.getReturnValue()

Returns the contents of the expectedReturn column in the current lookup table row.

Syntax

```
public java.lang.String getReturnValue()
```

Return Value

The contents of the Return Value column in the current lookup table row.

Exceptions

This method throws the following exception:

- `java.lang.Exception`. Reports problems attempting to retrieve the contents of the Return Value column.

If you handle this exception and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass Result for the script.

Example

For an example of this method, see *LookUpTable Example* on page 105.

LookUpTable.getValue()

Returns the contents of the specified column in the current lookup table row.

Syntax

```
public java.lang.String getValue(java.lang.String colName)
```

Element	Description
colName	The name of the column containing the value to retrieve.

Return Value

The contents of the specified column in the current lookup table row.

An auto-generated lookup table contains an input column for each parameter and two output columns — expectedReturn and expectedException. If you use additional output columns in a lookup table, you can use `getValue()` to retrieve values from those additional output columns.

Exceptions

This method throws the following exception:

- `java.lang.Exception`. Reports problems attempting to retrieve the contents of the specified column.

If you handle this exception and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass Result for the script.

LookUpTable.open()

Opens the specified lookup table.

Syntax

```
public void open(java.lang.String tablename)
```

Element	**Description**
`tablename`	The name of the lookup table to open.

Exceptions

This method throws the following exception:

- `java.lang.Exception`. Reports problems attempting to open the specified lookup table.

If you handle this exception and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass Result for the script.

Comments

Only one lookup table can exist for a given instance of the `LookUpTable` class.

Example

For an example of this method, see *LookUpTable Example* on page 105.

TestLog Class

This class lets you log information from test scripts and stubs.

The `TestLog` class extends the class `com.rational.test.tss.TSSLog`.

Overview

```
public class TestLog

com.rational.test.ct.TestLog
```

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this class.

TestLog Example

The following sample code uses the `writeException()` method to report an exception with a datapool operation. The example also uses the `message()` method from the extended `TssLog` class to log a number of status messages at various stages of the datapool operation. You can find examples of other `TestLog` methods in the section *LookUpTable Example* on page 105.

```
public void testMain(String[] args) {
   boolean fRetval = false;
   TSSDatapool dp = new TSSDatapool();
   int iDPCount = 0;
   try
   {
      // Initialize test services
      tms.startTestServices();

      // Initialize arguments, to java.math.BigDecimal
      // getBalance(longaccountID, java.lang.StringacctType) method
      long accountID = 0;
      java.lang.String acctType = null;
      java.math.BigDecimal expectedReturn = null;

      String sExpectedException = "";

      // Contact the bean home through JNDI
      InitialContext initContext = getInitialContext();
      Object o = initContext.lookup("ExecuteTransaction");
      ExecuteTransactionHome home = (ExecuteTransactionHome)
         PortableRemoteObject.narrow(o, ExecuteTransactionHome.class);
      // Declare arguments to the create method

      // Initialize arguments for the create method


      // Invoke the create method.
      ExecuteTransaction remote = home.create();
/**
      // Modify below to use a finder method and comment out the above
      // remote creation code.
      <keyvalue_declaration: Error>
      <createparam_init: Error>
      ExecuteTransactionKey key = new
           ExecuteTransactionKey(<create_params: Error>)
           ExecuteTransaction remote = home.findByPrimaryKey(key);
**/

      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();
         try
         {
            // Test java.math.BigDecimal getBalance(longaccountID,
            // java.lang.StringacctType)
            // java.math.BigDecimal actualReturn =
            // remote.getBalance(accountID, acctType);
            java.math.BigDecimal actualReturn = null;
            actualReturn =  remote.getBalance(accountID, acctType);

            if (sExpectedException.equals(""))
            {
               if (expectedReturn.equals(actualReturn))
                  TestLog.message("Expected result",
                     TSS_LOG_RESULT_PASS,
                     "Call to getBalance returned expected value");
               else
                  TestLog.message("Unexpected result",
                     TSS_LOG_RESULT_FAIL,
                     "Call to getBalance returned unexpected value, " +
                     ( (actualReturn) ).toString() + ".");
            }
            else
            {
               TestLog.message("Unexpected result",
                  TSS_LOG_RESULT_FAIL, "Expected exception,
                  " + sExpectedException + " was not thrown.");
            }
         }
         catch (Exception e)
         {
            if (e.getClass().getName().equals(sExpectedException))
            {
               // Expected exception occurred.  Log success.
               TestLog.message("Expected result", TSS_LOG_RESULT_PASS,
                  "Expected exception, "  + sExpectedException
                  + " was thrown.");
            }
            else
            {
               TestLog.message("Unexpected result",TSS_LOG_RESULT_FAIL,
                  "Unexpected exception, " + e.getClass().getName()
                  + " was thrown.");
            }
         }
         fRetval = dp.fetch();
      }

      if (iDPCount == 0)
      {
         // Datapool did not contain any rows.  Log a warning.
         TestLog.message("Empty Datapool", TSS_LOG_RESULT_WARN,
               "Datapool, " + sDPName + "is empty.");
      }
   }
   catch (Exception e)
   {
      TestLog.writeException(e);
   }
   finally
   {
      dp.close();
      tms.endTestServices();
   }
}
}
```

Summary

This class contains the following methods:

Method	**Description**
`writeException()`	Lets you log an exception that was thrown in a test script.
`writeStubException()`	Lets you log information about an exception that was thrown during the execution of a Rational QualityArchitect stub.
`writeStubMessage()`	Lets you log a message relating to the execution of a Rational QualityArchitect stub.

Note: In addition to these methods, you can also use the methods in the `TSSLog` class, as summarized in section *Logging Class* on page 32.

Constructor

Syntax

```
public TestLog()
```

TestLog.writeException()

Lets you log an exception that was thrown in a test script.

Syntax

```
public static boolean writeException(java.lang.Exception e)
```

Element	**Description**
e	The exception to log.

Return Value

`true` if the log attempt was successful, and `false` if the log attempt failed.

Comments

This method logs a Fail result for the test script.

Example

For an example of this method, see *TestLog Example* on page 112.

TestLog.writeStubException()

Lets you log information about an exception that was thrown during the execution of a Rational QualityArchitect stub.

Syntax

```
public static void writeStubException(java.lang.String
   description, java.lang.Exception e)
```

Element	Description
description	A description of the exception.
e	The exception to log.

Exceptions

This method throws the following exception:

- `java.lang.Exception`. Reports problems attempting to write a stub exception to the log.

If you handle this exception and do not log it, TestManager will not be aware of the exception and will not log a Fail result for it. The script will continue to run, and TestManager could log a Pass Result for the script.

Comments

The description appears in the **Description** field of the Log Event Properties dialog box.

Example

For examples of this method, see *LookUpTable Example* on page 105.

TestLog.writeStubMessage()

Lets you log a message relating to the execution of a Rational QualityArchitect stub, and also a description of the message.

Syntax

```
public static void writeStubMessage(java.lang.String message,
   java.lang.String description)
```

Element	Description
message	The message to insert into the log.
description	A description of the message. The description lets you expand upon the logged message.

Comments

The message appears in the **Log Event** column of the LogViewer. The description appears in the **Description** field of the Log Event Properties dialog box.

Example

For examples of this method, see *LookUpTable Example* on page 105.

TestLog.writeStubMessage()

Verification Services 5

Introduction to Verification Points

This chapter describes verification points and provides the basic concepts involved in adding verification points to test scripts. The chapter contains the following topics:

For information about creating a new verification point type, see *Implementing a New Verification Point* on page 161.

About Verification Points

A *verification point* is a mechanism for testing, or *verifying*, the behavior of the component-under-test.

Using Rational QualityArchitect, you can verify return values, the values of input/output parameters, and side effects — that is, how the behavior of the component-under-test affects the component itself as well as other objects. For example, in a banking application, you might want to verify that a component correctly calculates a monthly mortgage payment for a given set of inputs such as loan amount, interest rate, and life of loan.

You establish verification points in your test scripts using the classes and interfaces provided in the `com.rational.test.vp` package. You can use the contents of this package in two ways:

- To verify data in a JDBC datasource, use the `Database...` classes. These are the classes you typically use when recording or writing scripts for EJB testing.

  For details, see *Database Verification Point Reference* on page 131.

- To perform any other type of automated verification, you must first implement a new verification point type. For example, if you want to verify the properties of an object, you must first implement classes that capture, encapsulate, and compare the object's properties. A verification point implementer implements verification point classes based on the abstract verification point framework provided in the `com.rational.test.vp` package.

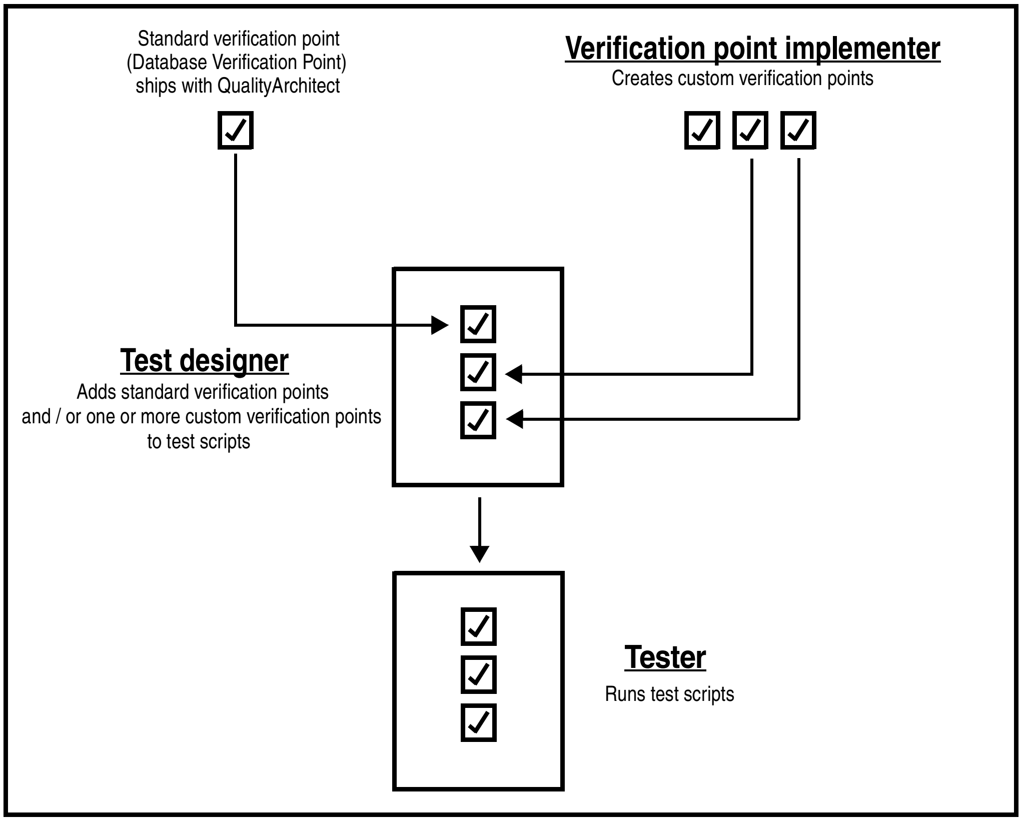
  For details, see *Verification Point Framework Reference* on page 181.

Roles in Working with Verification Points

The following testing team members use the `com.rational.test.vp` package and its documentation. Depending upon the requirements of your site, the same person or different persons perform the different tasks.

- The verification point *implementer* implements new verification points based on the verification point framework described in *Verification Point Framework Reference* on page 181.
- The *test designer* writes the scripts used for testing a component-under-test. In component testing, test designers incorporate existing verification point types into their test scripts — that is, the database verification point provided with Rational QualityArchitect plus any verification point types created by the verification point implementer.
- The *tester* runs the test scripts that the test designer writes.

The following diagram illustrates the different roles of the test team:


Standard verification point (Database Verification Point) ships with QualityArchitect
Verification point implementer
Creates custom verification points
Test designer
Adds standard verification points and / or one or more custom verification points to test scripts
Tester
Runs test scripts

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.

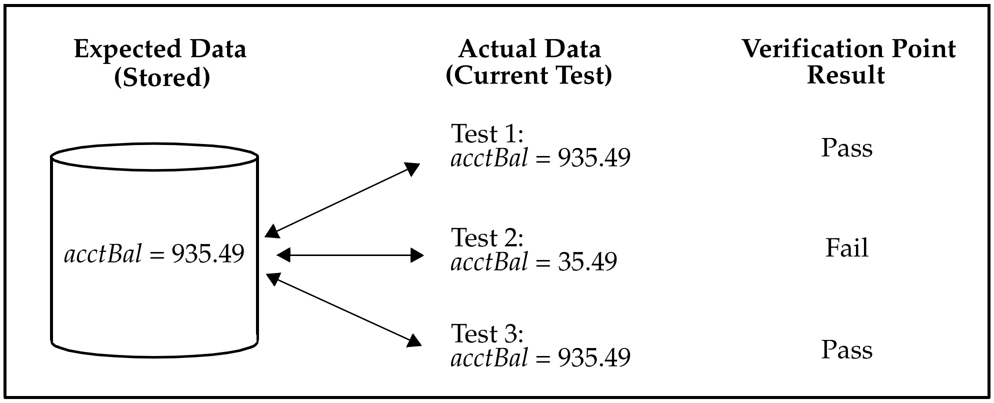
  Expected data can be data that is explicitly specified (for example, a person's name, social security number, or account number), or data that is the result of some calculation (for example, a monthly mortgage payment resulting from inputs of loan amount, interest rate, and number of payments).

- Data whose validity is unknown and must be verified.

  This data is always 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 logged automatically.

  Note: If the test script sets the `OPTION_EXPECT_FAILURE` option through the constructor or through the `setOptions()` method of the specialized Verification Point class, the verification point passes only if the data comparison fails.

In the following figure, the account balance 935.49 is the expected data for a given input (an account number). In three subsequent tests, the stored expected data is compared against the actual data captured during each test. In this example, the verification point passes if the expected data matches the actual data:



Types of Verification Points

The verification point framework provides for three types of verification points:

- Static
- Dynamic
- Manual

The following table summarizes the differences between verification point types:

	Expected Data Object	Actual Data Object
Static Verification Point	Captured when script is first run.	Captured at subsequent script runs.
Dynamic Verification Point	Test script passes to verification point.	Captured at script runtime.
Manual Verification Point	Test script passes to verification point.	Test script passes to verification point.

Static Verification Points

Static verification points are regression-style tests — in other words, the successful operation of the component-under-test is implicitly defined by the component's state during an earlier running of the test script, when the captured data was known to be correct.

With static verification points, the expected data object is captured during the first execution of the test script and is saved in the datastore as the baseline for subsequent executions of the test script. The expected data remains persistent unless and until it is explicitly replaced with new expected data. (To insert new expected data, click **File > Replace Baseline with Actual** in the Grid Comparator.)

Each subsequent time the test script is run, an actual data object is captured from the component-under-test. The expected data object is retrieved from the datastore and compared with the actual data captured in the current test run. The results are logged automatically.

Dynamic Verification Points

Dynamic verification points differ from static verification points in that, with dynamic verification points, you, the test script author, explicitly define the successful operation of the component-under-test, rather than implicitly defining it by a previous state of the component-under-test.

With dynamic verification points, the expected data object is passed to the verification point at test runtime. The expected data object is not retrieved from the datastore after having been captured in an earlier execution of the test script, nor is it managed in any way by the verification point framework, as is the case with static verification points.

How the expected data is passed to a verification point is up to you as the author of the test script. For example, you might hard-code the data into the script, supply the data through a datapool, or read the data from a Java properties file.

When a dynamic verification point is executed, the expected data object is passed as a parameter to the verification point's `performTest()` method. The verification point then captures the actual data object from the component-under-test, compares the expected and actual data objects, and automatically logs the results.

Manual Verification Points

With manual verification points, both the expected and actual data objects are passed to the verification point's `performTest()` method at test runtime. Expected and actual data objects are not provided by the verification point framework, as is the case with static verification points (where the framework provides both expected and actual data objects) and dynamic verification points (where the framework provides actual data objects only).

In other words, with manual verification points, you as the test designer are responsible for providing both the expected and the actual data objects. This frees you from relying on the framework's `VerificationPointDataProvider` class to construct objects, allowing you to construct your own objects. The framework simply compares the data objects you provide and logs the results.

Verification Point Framework

The `com.rational.test.vp` package includes the pre-defined database verification point for verifying data in a JDBC database. This is typically the verification point you use in writing scripts for EJB testing.

If you need to use other kinds of verification points, the verification point implementer must first extend and implement the class and interfaces in the verification point framework provided in the `com.rational.test.vp` package.

The verification point framework contains the following class and interfaces:

- `VerificationPoint` class
- `VerificationPointData` interface
- `VerificationPointDataProvider` interface
- `VerificationPointDataRenderer` interface
- `VerificationPointComparator` interface

For details about the framework, see Chapter 8, *Verification Point Framework Reference*.

Verification Point Classes

Conceptually, a verification point is made up of the following five classes:

- A Verification Point class, which extends the framework's `VerificationPoint` abstract class.

  This class contains the verification point's *metadata* — that is, the information that determines the data to capture for this verification point. Examples of verification point metadata include the list of properties for a user-defined object properties verification point, or connection information and SELECT statements for the JDBC database verification point that is included in this package. This class is also responsible for implementing its own serialization. By requiring your specific verification point implementations to perform their own serialization, you can support all file formats (such as INI, XML, and standard Java serialization).

- A Verification Point Data class, which implements the framework's `VerificationPointData` interface.

  This class encapsulates and serializes a single snapshot of either expected or actual data. An instance of this class can be populated by the `captureData()` method of a `VerificationPointDataProvider` class, or it can be populated manually in the test script — for example, by literal values or by values from a datapool. Each implementation of the `VerificationPointData` interface is required to provide its own serialization methods, once again for support of all possible file formats.

  Note: For the current Rational QualityArchitect release, Verification Point Data classes must serialize to a .CSV file format. This restriction will be removed in a future release of Rational QualityArchitect.

- A Verification Point Data Provider class, which implements the framework's `VerificationPointDataProvider` interface.

  This class is a pluggable link between a Verification Point class (which defines a verification point's metadata) and a Verification Point Data class (which stores data for a verification point). Specifically, this class implements the `captureData()` method to populate a Verification Point Data object for a given Verification Point object.

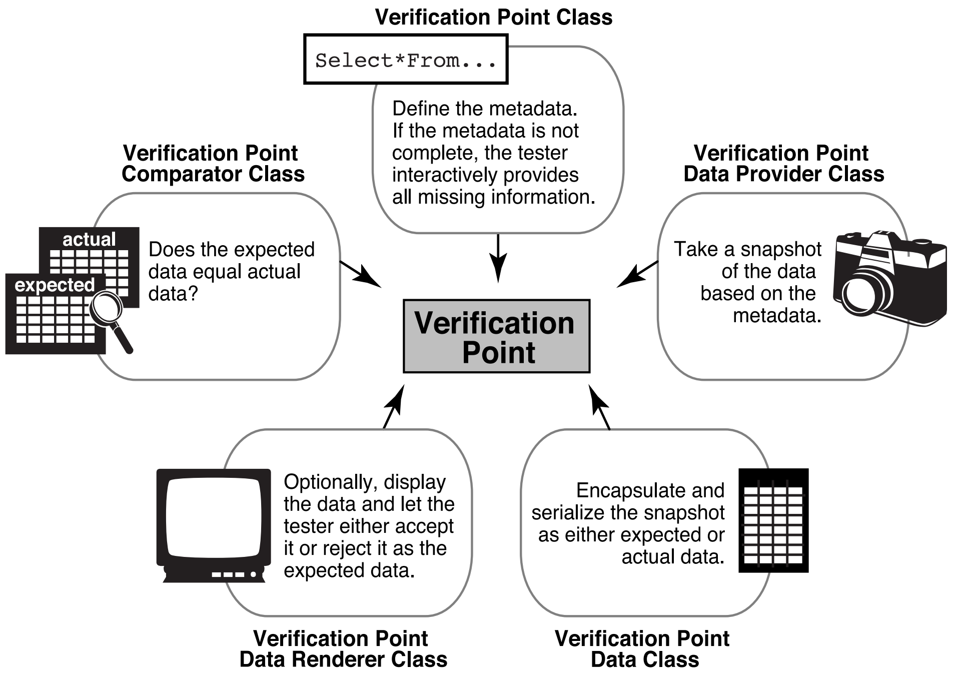
- A Verification Point Data Renderer class, which implements the framework's `VerificationPointDataRenderer` interface.

This class provides the capability of displaying the data stored in the Verification Point Data class, allowing the tester to interactively accept or reject that data as a baseline for a static verification point. To enable this capability, the test designer specifies the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method of the Verification Point class being implemented.

- A Verification Point Comparator class, which implements the framework's `VerificationPointComparator` interface.

  This class provides a method to compare two `VerificationPointData` objects and determine if the comparison succeeds or fails. The comparison can test for equality between the expected and actual data, or it can test for some other condition (for example, that the actual data falls within a given range).

  The following figure summarizes the verification point classes:


Verification Point Class
Select*From...
Define the metadata. If the metadata is not complete, the tester interactively provides all missing information.
Verification Point Comparator Class
actual
expected
Does the expected data equal actual data?
Verification Point Data Provider Class
Take a snapshot of the data based on the metadata.
Verification Point
Optionally, display the data and let the tester either accept it or reject it as the expected data.
Encapsulate and serialize the snapshot as either expected or actual data.
Verification Point Data Renderer Class
Verification Point Data Class

Setting Up Verification Points in Test Scripts

This section outlines the actions that you, the test designer, need to take to set up a verification point in a test script.

Use the following actions outlined as a guideline for setting up a verification point. You may need to perform other actions to accommodate the requirements of a particular verification point implementation.

Note that the verification point framework does much of the work that is required to perform a verification.

Setting Up a Static Verification Point

To set up a static verification point:

1. Specify the metadata for the verification point.
2. Execute the verification point.

The following sections provide information to help you perform these steps.

Step 1. Specify the Metadata for the Verification Point

The specialized `VerificationPoint` class encapsulates a verification point's metadata. Metadata includes the following kinds of information:

- Information that defines the kind of data that you want to capture and test. Here are two examples of this type of metadata:
  - With the pre-defined database verification point, the SQL statement that retrieves data from a JDBC data source. (For information about the database verification point, see Chapter 6, *Database Verification Point Reference*.)
  - If you are testing the properties of a component, the names of the particular properties to capture.
- Information needed to access the source of the data to capture (such as information used to connect to a JDBC data source).
- Possibly, one or more verification point options, such as whether to require case-sensitive matches of string data.

Verification point metadata can be specified either explicitly or implicitly:

- Metadata that is specified *explicitly* in the test script is either passed in as parameters to the constructor of the specialized `VerificationPoint` class, or it is specified through user-defined `set...` methods in the specialized `VerificationPoint` class.

Verification points that you record using the Rational QualityArchitect Session Recorder or that you generate through a Rational Rose model are explicitly defined — that is, the metadata is automatically hard-coded to the constructor.

Note: Because explicitly provided metadata can be assigned to test script variables, you can use datapools to supply metadata information to your test scripts.

- *Implicitly* defined metadata is specified in either of the following ways:
  - If a verification point's metadata is not fully specified when the verification point is executed for the first time, the framework invokes the `defineVPcallback()` method. This method runs a user-defined UI that prompts the tester for the metadata information. (The UI is typically developed by the verification point implementer.) After the metadata is captured, the framework writes the metadata to the datastore.
  - In subsequent executions of the verification point, the framework retrieves the metadata from the datastore and uses it as the metadata for the verification point.

  Note: Because implicitly provided metadata is retrieved from the datastore rather than being assigned to test script variables, you cannot use datapools with this type of metadata.

For more information about how to provide verification point metadata, see *VerificationPoint Class* on page 182.

Step 2. Execute the Verification Point

To execute a verification point, call the `performTest()` method in the specialized `VerificationPoint` class, as follows:

- If the verification point operates on a component within your test script's scope, pass that object to the `performTest()` method.
- If the verification point operates on an external object (such as a deployed EJB or a recordset in a database), pass `null` to the `performTest()` method.

Using the metadata in the specialized `VerificationPoint` class, the framework captures the actual data for the test. The framework also checks the datastore for an expected (baseline) data object to compare against the actual data:

- If the expected data object exists, the framework compares the expected data object with the actual data object, and then logs the result.
- If no expected data object exists, the framework attempts to store the captured data as a baseline for future executions of the verification point.

However, if no expected data object exists and you have included the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method, the framework first invokes an implementer-defined UI that prompts the tester to verify that the captured data is correct.

If the tester accepts the displayed data as being correct, the framework stores the data object in the datastore as the expected data for subsequent tests. If the tester rejects the displayed data, the framework logs an error, and verification point execution ends. No expected data object is stored.

For an example of a static verification point setup in a test script, see *Example of a Static Database Verification Point* on page 132.

Setting Up a Dynamic Verification Point

Setting up a dynamic verification point is similar to setting up a static verification point. However, before the test script executes the verification point, the test script must create the expected data object. The framework is responsible for capturing and building the actual data object, just as it does for a static verification point.

You create the expected data object using the appropriate implementation of the `VerificationPointData` interface.

After the expected data object is created, you can pass it to the `performTest()` method when you execute the verification point.

For an example of a dynamic verification point setup in a test script, see *Example of a Dynamic Database Verification Point* on page 133.

Setting Up a Manual Verification Point

Setting up a manual verification point is similar to setting up a static verification point. However, before the test script executes the verification point, the test script must create both the expected and actual data objects.

You create the expected and actual data objects using the appropriate implementation of the `VerificationPointData` interface.

After the expected and actual data objects are created, you can pass them to the `performTest()` method when you execute the verification point.

Database Verification Point Reference 6

About the Database Verification Point

A *database verification point* is a pre-constructed verification point used to verify data in a JDBC accessible data source. This is the verification point that you typically use in EJB testing.

You can use this verification point within a test script to ensure that the changes that the component-under-test makes to the data source are correct.

Requirements for Using the Database Verification Point

The database verification point requires Rational QualityArchitect.

In addition, the CLASSPATH must reference a number of .JAR files. For a list of the required .JAR files, see *Running Test Scripts* on page 8.

Components of the Database Verification Point

The database verification point contains the following classes and interface:

- *DatabaseVP Class* on page 134
- *DatabaseVPComparator Class* on page 145
- *DatabaseVPData Class* on page 147
- *DatabaseVPDataProvider Class* on page 153
- *DatabaseVPDataRenderer Class* on page 155
- *DataTable Interface* on page 157

These classes are included in the package `com.rational.test.vp`.

Examples

This section contains examples of how you can insert a static and a dynamic database verification point into a test script.

Note that the verification point framework does much of the work for you. The test script defines the verification point's metadata and calls the `performTest()` method in the specialized Verification Point class. Depending on whether you are inserting a static, dynamic, or manual verification point, the test script might also build the expected data object and the actual data object.

For an overview of the steps required to insert a verification point into a script, see *Setting Up Verification Points in Test Scripts* on page 127.

Example of a Static Database Verification Point

In a static verification point, the `performTest()` method does not pass data objects to the verification point . As a result, the framework must provide both the expected (baseline) and actual data objects.

```
String sJDBCdriver = "sun.jdbc.odbc.JdbcOdbcDriver";
String sJDBCurl = "jdbc:odbc:COFFEEBREAK";
String sJDBCuser = "";
String sJDBCpassword = "";

DatabaseVP regressionVP = new DatabaseVP( "RegressionVP1",
          "SELECT * FROM COFFEES",sJDBCuser, sJDBCpassword,
          sJDBCdriver, sJDBCurl );

regressionVP.performTest( null );
```

Example of a Dynamic Database Verification Point

In a dynamic verification point, the test script creates a `DatabaseVPData` object for the expected data and passes the expected data object to the verification point through the `performTest()` method. As a result, the framework encapsulates only the actual data object.

```
String sJDBCdriver = "sun.jdbc.odbc.JdbcOdbcDriver";
String sJDBCurl = "jdbc:odbc:COFFEEBREAK";
String sJDBCuser = "";
String sJDBCpassword = "";
String sFilter = "1";

DatabaseVPData vpdExpected = new DatabaseVPData();
String[] asColumns = new String[2];
asColumns[0] = "Brand";
asColumns[1] = "Price";
vpdExpected.setColumns(asColumns);

Vector vData = new Vector();
String[] asData = new String[2];
asData[0] = "Peets";
asData[1] = "5.5";
vData.add(asData);
vpdExpected.setData(vData);

String sSQL = "SELECT Brand, Price FROM COFFEES WHERE ID = " + sFilter;
DatabaseVP VP1 = new DatabaseVP( "CoffeeVp1", sSQL, sJDBCuser,
   sJDBCpassword, sJDBCdriver, sJDBCurl );

// Perform the test
VP1.performTest(null, vpdExpected);
```

DatabaseVP Class

This class implements a database verification point.

The `DatabaseVP` object contains the metadata needed for encapsulating data in a `DatabaseVPData` object — namely:

- The SELECT statement for retrieving data from the target data source.
- A valid JDBC user name (if none, an empty string).
- The valid JDBC password for the user name (if none, an empty string).
- The JDBC driver for the target data source.
- The JDBC URL for the target data source.

In addition, the `DatabaseVP` object contains the database verification point name. It also contains options for affecting the behavior of the verification point.

To execute the database verification point, call the `performTest()` method in this class (inherited from the `VerificationPoint` class).

Overview

```
public class DatabaseVP
extends com.rational.test.vp.VerificationPoint

java.lang.Object
  |
  +--com.rational.test.vp.VerificationPoint
        |
        +--com.rational.test.vp.DatabaseVP
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

Summary

This class contains the following field:

Field	Description
`OPTION_TRIM`	`static int`. Specifies that values captured from the `DatabaseVP` should have whitespace trimmed from the right and left sides.

Fields Inherited from the VerificationPoint Class
`bIsDefined`, `bIsValid`, `COMPARE_CASEINSENSITIVE`, `COMPARE_CASESENSITIVE`, `OPTION_EXPECT_FAILURE`, `OPTION_USER_ACKNOWLEDGE_BASELINE`, `sFailureDescription`, `VERIFICATION_ERROR`, `VERIFICATION_FAILED`, `VERIFICATION_NO_RESULT`, `VERIFICATION_SUCCEEDED`

Note: To turn on multiple options, use the `OR` (|) operator. To remove an option after you have set it, but leave all other options unchanged, use the `AND` (&) and `NOT` (~) operators. The following are examples of turning options on and off:

- Turn two options on:

  ```
  MyVP.setOptions(OPTION_TRIM | OPTION_EXPECT_FAILURE);
  ```

- Turn off the `OPTION_TRIM` option, but leaves all other options unchanged:

  ```
  MyVP.setOptions(MyVP.Options & (~OPTION_TRIM));
  ```

This class contains the following methods:

Method	Description
`getCon()`	Retrieves the current connection object used to connect to the JDBC data source.
`getJDBCdriver()`	Retrieves the current driver used in the connection to the JDBC data source.
`getJDBCpassword()`	Retrieves the current password for connecting to the JDBC data source.
`getJDBCurl()`	Retrieves the current URL used to connect to the JDBC data source.
`getJDBCuser()`	Retrieves the current user ID for connecting to the JDBC data source.

Method	Description
`getSQL()`	Retrieves the current SQL statement used to capture data from the JDBC data source.
`getStmt()`	Retrieves the current JDBC statement.
`readFile()`	Deserializes a verification point object from the specified InputStream.
`setCon()`	Sets the connection object for the JDBC data source.
`setJDBCdriver()`	Sets the JDBC driver used to connect to the JDBC data source.
`setJDBCpassword()`	Sets the password for the connection to the JDBC data source.
`setJDBCurl()`	Sets the JDBC URL used in the connection to the JDBC data source.
`setJDBCuser()`	Sets the user ID for the connection to the JDBC data source.
`setSQL()`	Sets the SQL statement to use in capturing data from the JDBC data source.
`setStmt()`	Sets the JDBC statement.
`writeFile()`	Serializes the verification point object to the specified OutputStream.

Methods Inherited from the VerificationPoint Class
`codeFactory_getPrefix, codeFactory_setPrefix, getIsDefined, getLog, getLogActualFile, getLogBaselineFile, getLogMetaFile, getMasterBaselineFile, getMasterMetaFile, getOptions, getVPname, initializeVP, performTest, performTest, performTest, setIsDefined, setOptions, setVPname`

Constructor

The constructor takes one of three forms, depending on the parameters passed to it:

Syntax 1

This constructor specifies only the name of the verification point. If you execute the verification point before specifying its metadata, the tester is prompted to specify the verification point's metadata. The metadata includes JDBC connection information and a SQL statement to capture the data to test.

```
public DatabaseVP(java.lang.String sVPname)
```

Element	Description
sVPname	The name of the verification point (40 characters maximum).

Syntax 2

This constructor specifies the name of the verification point plus the verification point's metadata.

```
public DatabaseVP(java.lang.String sVPname, java.lang.String
   sSQL, java.lang.String sJDBCuser, java.lang.String
   sJDBCpassword, java.lang.String sJDBCdriver,
   java.lang.String sJDBCurl)
```

Element	Description
sVPname	The name of the verification point (40 characters maximum).
sSQL	The select statement that this `DatabaseVP` uses to capture data from the data source.
sJDBCVuser	The JDBC user name.
sJDBCpassword	The JDBC password for the user.
sJDBCdriver	The Java class for the JDBC driver for this data source.
sJDBCurl	The URL specifying the target JDBC data source.

Syntax 3

This constructor specifies the name of the verification point, the verification point's metadata, and any options that customize the behavior of the verification point.

```
public DatabaseVP(java.lang.String sVPname, java.lang.String
   sSQL, java.lang.String sJDBCuser, java.lang.String
   sJDBCpassword, java.lang.String sJDBCdriver,
   java.lang.String sJDBCurl, int iOptions)
```

Element	Description
sVPname	The name of the verification point (40 characters maximum).
sSql	The select statement that this `DatabaseVP` uses to capture data from the data source.
sJDBCuser	The JDBC user name.
sJDBCpassword	The JDBC password for the user.
sJDBCdriver	The Java class for the JDBC driver for this data source.
sJDBCurl	The URL specifying the target JDBC data source.
iOptions	A bitfield of options that customize the behavior of this verification point. Options can include the following pre-defined options and any user-defined options: ▪ `OPTION_TRIM` ▪ The following options inherited from `VerificationPoint`: ❑ `COMPARE_CASESENSITIVE` ❑ `COMPARE_CASEINSENSITIVE` ❑ `OPTION_USER_ACKNOWLEDGE_BASELINE` ❑ `OPTION_EXPECT_FAILURE` `COMPARE_CASESENSITIVE` is the default.

DatabaseVP.getCon()

Retrieves the current connection object used to connect to the JDBC data source.

Syntax

```
public java.sql.Connection getCon()
```

Return Value

The current connection object.

DatabaseVP.getJDBCdriver()

Retrieves the current driver used in the connection to the JDBC data source.

Syntax

```
public java.lang.String getJDBCdriver()
```

Return Value

The current JDBC driver.

DatabaseVP.getJDBCpassword()

Retrieves the current password for connecting to the JDBC data source.

Syntax

```
public java.lang.String getJDBCpassword()
```

Returns Value

The current password.

DatabaseVP.getJDBCurl()

Retrieves the current URL used to connect to the JDBC data source.

Syntax

```
public java.lang.String getJDBCurl()
```

Return Value

The current URL.

DatabaseVP.getJDBCuser()

Retrieves the current user ID for connecting to the JDBC data source.

Syntax

```
public java.lang.String getJDBCuser()
```

Return Value

The current user ID.

DatabaseVP.getSQL()

Retrieves the current SQL statement used to capture data from the JDBC data source.

Syntax

```
public java.lang.String getSQL()
```

Return Value

The current SQL statement.

DatabaseVP.getStmt()

Retrieves the current JDBC statement.

Syntax

```
public java.sql.Statement getStmt()
```

Return Value

The current JDBC statement.

DatabaseVP.readFile()

Deserializes a verification point object from the specified InputStream.

Syntax

```
public void readFile(java.io.InputStream in)
```

Element	**Description**
in	The InputStream from which the object is read.

Exceptions

This method throws the following exception:

- `java.io.IOException`. An error has occurred in attempting to read from the InputStream.

Comments

This method implements `readFile()` in the `VerificationPoint` class.

DatabaseVP.setCon()

Sets the connection object for the JDBC data source.

Syntax

```
public void setCon(java.sql.Connection con)
```

Element	**Description**
con	The connection object to use in connecting to the JDBC data source.

Comments

If *con* is not provided, a new object is created by the database verification point, as necessary.

DatabaseVP.setJDBCdriver()

Sets the JDBC driver used to connect to the JDBC data source.

Syntax

```
public void setJDBCdriver(java.lang.String sJDBCdriver)
```

Element	Description
sJDBCdriver	The driver used to connect to the JDBC data source.

DatabaseVP.setJDBCpassword()

Sets the password for the connection to the JDBC data source.

Syntax

```
public void setJDBCpassword(java.lang.String sJDBCpassword)
```

Element	Description
sJDBCpassword	The password for connecting to the JDBC data source.

DatabaseVP.setJDBCurl()

Sets the JDBC URL used in the connection to the JDBC data source.

Syntax

```
public void setJDBCurl(java.lang.String sJDBCurl)
```

Element	Description
sJDBCurl	The URL used in the connection to the JDBC data source.

DatabaseVP.setJDBCuser()

Sets the user ID for the connection to the JDBC data source.

Syntax

```
public void setJDBCuser(java.lang.String sJDBCuser)
```

Element	Description
sJDBCuser	The user ID for connecting to the JDBC data source.

DatabaseVP.setSQL()

Sets the SQL statement to use in capturing data from the JDBC data source.

Syntax

```
public void setSQL(java.lang.String sSQL)
```

Element	Description
sSQL	The SQL statement to use.

DatabaseVP.setStmt()

Sets the JDBC statement.

Syntax

```
public void setStmt(java.sql.Statement stmt)
```

Element	Description
stmt	The JDBC statement.

Comments

If `stmt` is not provided, a new object is created by the database verification point, as necessary.

DatabaseVP.writeFile()

Serializes the verification point object to the specified OutputStream.

Syntax

```
public void writeFile(java.io.OutputStream out)
```

Element	Description
out	The OutputStream to which the object is written.

Exceptions

This method throws the following exception:

- `java.io.IOException`. An error has occurred in attempting to write to the OutputStream.

Comments

Metafile format is used so that the Rational comparators can read the file. For information, see *Step 5. Provide Serialization Services for the Metadata* on page 168.

This method implements `writeFile()` in the `VerificationPoint` class.

DatabaseVPComparator Class

The verification point framework calls the `compare()` method in this class to compare two `DatabaseVPData` objects. The comparison is for either case-sensitive equality or case-insensitive equality, depending on the options set in the `DatabaseVP` object that is driving the comparison.

Overview

```
public class DatabaseVPComparator
extends java.lang.Object
implements com.rational.test.vp.VerificationPointComparator

java.lang.Object
  |
  +--com.rational.test.vp.DatabaseVPComparator
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

Summary

This class contains the following method:

Method	Description
`compare()`	Compares an expected data object and an actual data object, both of type `VerificationPointData`, and determines whether the test succeeds or fails.

Methods Inherited from Class java.lang.Object
`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor

Syntax

```
public DatabaseVPComparator()
```

DatabaseVPComparator.compare()

Compares an expected data object and an actual data object and determines whether the test succeeds or fails.

Syntax

```
public boolean compare(VerificationPointData vpdExpected,
   VerificationPointData vpdActual, java.lang.Object
   objOptions, java.lang.StringBuffer sFailureDescription)
```

Element	Description
vpdExpected	The expected data object.
vpdActual	The actual data object.
objOptions	Options that are passed from the `DatabaseVP` class to qualify the comparison. Options can include the following pre-defined options, plus any user-defined options. `OPTION_TRIM`, `OPTION_EXPECT_FAILURE`, `COMPARE_CASESENSITIVE`, and `COMPARE_CASEINSENSITIVE` .
sFailureDescription	An output parameter that contains the differences between the expected and actual data objects in a failed verification point. The failure description is written to the log.

Return Value

A `boolean` value indicating whether the test passed or failed.

Comments

The expected and the actual data objects are `DatabaseVPData` implementations of `VerificationPointData`.

This method is specified by the `compare()` method in the interface `VerificationPointComparator`.

DatabaseVPData Class

This class encapsulates and serializes the data being verified by the database verification point.

The data that this class stores is conceptually just a recordset. The data is stored in two data constructs represented in the `DataTable` interface:

- Columns — An array of strings representing the column names in the recordset.
- Data — A `vector` of arrays of strings, with each array representing one row of data from the recordset.

If you want to build a `DatabaseVPData` object by hand in order to run a dynamic or manual verification point, you can do so by populating the `Columns` and `Data` objects using the `get...` and `set...` methods provided in this class.

You can find an example of a hand-built `DatabaseVPData` object in the section *Example of a Dynamic Database Verification Point* on page 133.

Overview

```
public class DatabaseVPData
extends java.lang.Object
implements com.rational.test.vp.VerificationPointData,
com.rational.test.vp.DataTable

java.lang.Object
  |
  +--com.rational.test.vp.DatabaseVPData
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

Summary

This class contains the following methods:

Method	**Description**
`getColumns()`	Retrieves the column names in the table.

Method	Description
`getData()`	Retrieves data from the table
`getFileExtension()`	Returns the extension of the file used to store the data object
`getNumCols()`	Retrieves the number of columns in the table.
`getNumRows()`	Retrieves the number of rows in the table.
`readFile()`	Reads the expected or actual data object from the specified InputStream.
`setColumns()`	Specifies the column names in the table.
`setData()`	Specifies the data in the table.
`writeFile()`	Writes the expected or actual data object to the specified OutputStream.

Methods Inherited from Class java.lang.Object
`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor

Syntax

```
public DatabaseVPData()
```

DatabaseVPData.getColumns()

Retrieves the column names in the table.

Syntax

```
public java.lang.String[] getColumns()
```

Comments

This method is specified by `getColumns()` in interface `DataTable`.

DatabaseVPData.getData()

Retrieves data from the table. Each array contains one row of data.

Syntax

```
public java.util.Vector getData()
```

Return Value

A `Vector` of arrays of strings, with each array representing one row of data from the recordset.

Comments

This method is specified by `getData()` in the `DataTable` interface.

DatabaseVPData.getFileExtension()

Returns the extension of the file used to store the data object.

Note: In the current release, CSV is the only supported file format. Other formats will be supported in future releases.

Syntax

```
public java.lang.String getFileExtension()
```

Return Value

The extension of the file used to store the data object.

Comments

The verification point framework uses the file extension to determine the format to use when it serializes files (for example, a CSV extension indicates a comma-separated-value text file).

This method is specified by `getFileExtension()` in the `VerificationPointData` interface.

DatabaseVPData.getNumCols()

Retrieves the number of columns in the table.

Syntax

```
public int getNumCols()
```

Return Value

The number of columns in the table.

Comments

This method is specified by `getNumCols()` in the `DataTable` interface.

DatabaseVPData.getNumRows()

Retrieves the number of rows in the table.

Syntax

```
public int getNumRows()
```

Return Value

The number of rows in the table.

Comments

This method is specified by `getNumRows()` in the `DataTable` interface.

DatabaseVPData.readFile()

Reads the expected or actual data object from the specified InputStream.

Syntax

```
public void readFile(java.io.InputStream in)
```

Element	**Description**
in	The InputStream from which the data is read.

Exceptions

This method throws the following exception:

- `java.io.IOException`. An input/output error has occurred.

Comments

This method is specified by `readFile()` in the `VerificationPointData` interface.

DatabaseVPData.setColumns()

Specifies the column names in the table.

Syntax

```
public void setColumns(java.lang.String[] asColumns)
```

Element	Description
asColumns	The array of the column names in the table.

Comments

This method is specified by `setColumns()` in the `DataTable` interface.

Example

For an example of this method, see *Example of a Dynamic Database Verification Point* on page 133.

DatabaseVPData.setData()

Specifies the data in the table. Each element in the `Vector` is an array of strings containing one row of data.

Syntax

```
public void setData(java.util.Vector vData)
```

Element	Description
vData	The data in the table.

Comments

This method is specified by `setData()` in the `DataTable` interface.

Example

For an example of this method, see *Example of a Dynamic Database Verification Point* on page 133.

DatabaseVPData.writeFile()

Writes the expected or actual data object to the specified OutputStream.

Syntax

```
public void writeFile(java.io.OutputStream out)
```

Element	Description
out	The OutputStream to which the object is written.

Exceptions

This method throws the following exception:

- `java.io.IOException`. An input/output error has occurred.

Comments

This method is specified by `writeFile()` in the `VerificationPointData` interface.

DatabaseVPDataProvider Class

This class provides the link between the `DatabaseVP` class and the `DatabaseVPData` class.

The `DatabaseVPDataProvider` class can create and populate a `DatabaseVPData` object based on the metadata in the `DatabaseVP` object. It does so by:

- Connecting to the database
- Creating a statement and connection (if necessary)
- Executing the specified SQL statement
- Building the `DatabaseVPData` object from the resulting recordset

This class is used with static verification points (for building expected and actual data objects) and with dynamic verification points (for building actual data objects only).

Overview

```
public class DatabaseVPDataProvider
extends java.lang.Object
implements com.rational.test.vp.VerificationPointDataProvider

java.lang.Object
  |
  +--com.rational.test.vp.DatabaseVPDataProvider
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

Summary

This class contains the following method:

Method	**Description**
`captureData()`	Builds an expected or actual data object of type `VerificationPointData`.

Methods Inherited from Class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor

Syntax

```
public DatabaseVPDataProvider()
```

DatabaseVPDataProvider.captureData()

iThis method builds a `VerificationPointData` object according to the metadata in the `VerificationPoint` class.

Syntax

```
public VerificationPointData captureData(java.lang.Object
   theObject, VerificationPoint VP)
```

Element	Description
theObject	For database verification points, pass `null` in this parameter.
VP	The Verification Point object that contains the verification point's metadata.

Return Value

An expected or actual data object.

Comments

This method is specified by `captureData()` in the interface `VerificationPointDataProvider`.

DatabaseVPDataRenderer Class

This class implements a renderer for any class that implements the `DataTable` interface.

The renderer creates a JDialog containing a scrollable JTable with the data from the supplied `DatabaseVPData` object.

If no expected data object exists in the datastore and the `OPTION_USER_ACKNOWLEDGE_BASELINE` option is set in the test script, the verification point framework invokes the `displayAndValidateData()` method in this class. This method lets the tester interactively accept or reject the displayed data as the baseline (expected) data for a static verification point.

Overview

```
public class DatabaseVPDataRenderer
extends java.lang.Object
implements com.rational.test.vp.VerificationPointDataRenderer

java.lang.Object
  |
  +--com.rational.test.vp.DatabaseVPDataRenderer
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

Summary

This class contains the following method:

Method	**Description**
`displayAndValidateData()`	Presents the tester with a visual representation of the data object as it exists before the expected (baseline) data is stored for this static verification point.

Methods Inherited from Class java.lang.Object
`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor

Syntax

```
public DatabaseVPDataRenderer()
```

DatabaseVPDataRenderer.displayAndValidateData()

Presents the tester with a visual representation of the data object as it exists before the expected (baseline) data is stored for this static verification point.

Syntax

```
public boolean displayAndValidateData(VerificationPointData
    vpdData)
```

Element	Description
vpdData	The data to present to the tester for confirmation.

Return Value

`true` if the tester accepts the displayed data, or `false` if the tester rejects the data.

Comments

This method is specified by `displayAndValidateData()` in the `VerificationPointDataRenderer` interface .

The verification point framework invokes this method when the following conditions exist:

- You have set the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method of the Verification Point class.
- No expected data object exists in the datastore when the test script calls the `performTest()` method of the Verification Point class for a static verification point.

When the method is invoked, it presents the tester with a visual representation of the data, and allows the tester to accept or reject the data:

- If the tester accepts the data, the framework adds the data to the datastore as the expected data for subsequent test runs.
- If the tester rejects the data, the framework logs the failure, and no expected data is stored for the verification point. The next time the tester runs the script, the framework again prompts the tester to accept the data.

DataTable Interface

This interface is implemented by a class that encapsulates a table of string data.

Overview

```
public interface DataTable
```

Applicability

Commonly used with Rational QualityArchitect.

This interface requires Rational QualityArchitect.

Summary

This class contains the following methods:

Method	**Description**
`getColumns()`	Retrieves the column names in the table.
`getData()`	Retrieves data from the table.
`getNumCols()`	Retrieves the number of columns in the table.
`getNumRows()`	Retrieves the number of rows in the table.
`setColumns()`	Specifies the column names in the table.
`setData()`	Specifies the data in the table.

DataTable.getColumns()

Retrieves the column names in the table.

Syntax

```
public java.lang.String[] getColumns()
```

Return Value

An array of the column names in the table.

DataTable.getData()

Retrieves data from the table.

Syntax

```
public java.util.Vector getData()
```

Return Value

A `vector` of arrays of data in the table. Each array represents one row of data from the recordset.

DataTable.getNumCols()

Retrieves the number of columns in the table.

Syntax

```
public int getNumCols()
```

Return Value

The number of columns in the table.

DataTable.getNumRows()

Retrieves the number of rows in the table.

Syntax

```
public int getNumRows()
```

Return Value

The number of rows in the table.

DataTable.setColumns()

Specifies the column names in the table.

Syntax

```
public void setColumns(java.lang.String[] asColumns)
```

Element	Description
asColumns	An array of the column names in the table.

DataTable.setData()

Specifies the data in the table. Each element in the `vector` is an array of strings containing one row of data.

Syntax

```
public void setData(java.util.Vector vData)
```

Element	Description
vData	The data in the table.

Implementing a New Verification Point 7

Introduction to Verification Point Implementation

The verification point framework is an open architecture that you can use to implement your own verification point types and execute them within the verification point framework.

This chapter describes the steps necessary to implement a new verification point type. It has the following topics:

- *Fundamentals for Implementing a Verification Point* on page 161 describes the components you must implement.
- *Integrating a Verification Point with QualityArchitect* on page 180 explains how your implemented components interact with the verification point framework and with the Rational QualityArchitect code generators to provide complete verification point services.

This chapter is intended only for implementers of new verification point types. If you are a test designer who is adding existing verification points to your scripts, you can skip this chapter. This chapter assumes a sound working knowledge of Java as well as an understanding of verification points.

Note: Some of the examples in this chapter use the `CTutil` class to retrieve values from an .ini file. If you want to refer to the `CTutil` class code for greater understanding of the examples, you can find the code in Appendix C, *CTutil Class Source Code*.

Fundamentals for Implementing a Verification Point

To implement a new verification point, you must implement the following classes:

- Verification Point (see page 162)
- Verification Point Data (see page 169)
- Verification Point Data Comparator (see page 174)

- Verification Point Data Provider (see page 176)
- Verification Point Data Renderer class (see page 179)

The following sections describe these classes.

Implementing the Verification Point Class

Your specialized Verification Point class must extend the `com.rational.test.vp.VerificationPoint` abstract class and implement all the abstract methods within it. For example, if you are implementing a verification point `DatabaseVP`, use the following code:

```
public class DatabaseVP extends com.rational.test.vp.VerificationPoint
```

Further, your Verification Point class inherits the framework's entire behavior from this abstract base class. For details about this inherited behavior, see *Integrating a Verification Point with QualityArchitect* on page 180.

Your specialized Verification Point class must perform the following tasks:

1 Define and maintain the metadata that describes the verification to be performed.

2 Supply a UI that allows a tester to specify the metadata.

3 Implement constructors that provide the new verification point's name and metadata.

4 Implement the Rational *code factory* methods. These framework methods automatically generate source code into a test script and are capable of creating instances of your verification point.

5 Provide serialization services for the verification point's metadata.

Step 1. Define and Maintain the Metadata

Your verification point must contain member variables and corresponding `get/set` methods for all attributes necessary to describe the verification point's metadata.

The following example illustrates the use of `get/set` methods for retrieving and assigning metadata such as a JDBC user ID, password, url, and a SQL statement:

```
private String sSQL = "";
private String sJDBCuser = "";
private String sJDBCpassword = "";
private String sJDBCdriver = "";
private String sJDBCurl = "";

public String getSQL() { return sSQL; }
public String getJDBCuser() { return sJDBCuser; }
public String getJDBCpassword() { return sJDBCpassword; }
public String getJDBCdriver() { return sJDBCdriver; }
public String getJDBCurl() { return sJDBCurl; }

public void setSQL( String sSQL ) { this.sSQL = sSQL; }
public void setJDBCuser( String sJDBCuser )
     { this.sJDBCuser = sJDBCuser; }
public void setJDBCpassword( String sJDBCpassword )
     { this.sJDBCpassword = sJDBCpassword; }
public void setJDBCdriver( String sJDBCdriver )
     { this.sJDBCdriver = sJDBCdriver; }
public void setJDBCurl( String sJDBCurl ) { this.sJDBCurl = sJDBCurl; }
```

Step 2. Supply a UI to Prompt for the Metadata

If a test script executes your verification point, but the verification point's metadata is not completely defined in the datastore, the verification point must run a UI that prompts the tester for the missing metadata. Specifically, you must provide the following features:

- The UI that prompts the user for the metadata.
- An implemented `defineVPcallback()` method. (This is an abstract method of the `VerificationPoint` base class.)

The `defineVPcallback()` method presents the tester with your UI that prompts for the metadata. When the metadata is retrieved, the method populates the verification point's member variables with the metadata values — for example:

```
public boolean defineVPcallback()
{
   // Invoke some UI and populate the class with the VP's definition.
}
```

Step 3. Implement the Constructors

Implement at least two constructors that use the `super` keyword to call the constructor of the `VerificationPoint` base class, as follows:

- One required constructor should have as its only parameter a string that specifies the name of your verification point. The QualityArchitect code generators (the Session Recorder and the Code Generator used with Rational Rose models) use this constructor to create an instance of the verification point class at code generation time.
- One required constructor with the following parameters:
  - A parameter that specifies the verification point name.
  - A parameter for each variable that contains a metadata value.

One of the tasks that the code factory methods (described in the next two steps) perform is to output code that invokes this constructor. As a result, this is the constructor that appears in scripts generated by a QualityArchitect code generator.

Both constructors must pass class objects for the following classes that you have implemented:

- Verification Point Data
- Verification Point Data Provider
- Verification Point Data Renderer
- Verification Point Data Comparator

The verification point framework can then create instances of these classes to store, serialize, capture, display, and compare the data on which your verification point operated. An example of creating instances of these classes to perform the above methods is shown as follows:

```
public DatabaseVP( String sVPname )
   {
      super(sVPname, DatabaseVPData.class,
         DatabaseVPDataProvider.class, DatabaseVPDataRenderer.class,
         DatabaseVPComparator.class);
      setIsDefined(false);
   }

public DatabaseVP( String sVPname, String sSQL, String sJDBCuser,
      String sJDBCpassword, String sJDBCdriver, String sJDBCurl )
   {
      super(sVPname, DatabaseVPData.class,
         DatabaseVPDataProvider.class, DatabaseVPDataRenderer.class,
         DatabaseVPComparator.class);
      this.sSQL = sSQL;
      this.sJDBCuser = sJDBCuser;
      this.sJDBCpassword = sJDBCpassword;
      this.sJDBCdriver = sJDBCdriver;
      this.sJDBCurl = sJDBCurl;

      if ( sSQL != null && !sSQL.equals("") && sJDBCdriver != null &&
         !sJDBCdriver.equals("") && sJDBCurl != null &&
         !sJDBCurl.equals(""))
      {
         setIsDefined(true);
      }
      else
      {
         setIsDefined(false);
      }
   }
```

Step 4. Implement the Code Factory Methods to Generate Code

The code factory methods are similar in function to Java Beans in that both provide additional design-time behavior that is integrated with a Java development environment.

If a QualityArchitect user wants to insert your verification point into a generated test script, the QualityArchitect code generator takes the following actions:

1. Creates an instance of the verification point (by calling the consructor that specifies just the verification point name).
2. Calls the `defineVPcallback()` method for the newly created verification point object, presenting the tester with the UI you created to prompt for the verification point's metadata.
3. After the tester specifies the metadata through the UI, the code generator invokes the code factory methods to produce Java source code. When inserted into the test script, this source code creates a verification point based on the metadata that the tester provided.

For information about how the code generators use the code factory methods, see *Integrating a Verification Point with QualityArchitect* on page 180.

To enable the code generators to insert an instance of your verification point into a test script, implement the following code factory methods:

- `codeFactory_getConstructorInvocation()` returns a string of Java code that calls the fully specified constructor of your verification point. Rather than hard-coding the metadata into the constructor call, you should externalize any variables that testers might want to supply with values from a datapool at test runtime.
- `codeFactory_getNumExternalizedInputs()`, called by the code generator, determines how many externalized input variables are present in the constructor call.
- `codeFactory_getExternalizedInputDecl()`, called by the code generator, retrieves each externalized metadata variable.

The code generators call the `codeFactory_getPrefix()` and `codeFactory_setPrefix()` methods; you are not required to call them. However, you must call `codeFactory_getPrefix()` when constructing the externalized variables returned by the `codeFactory_getConstructorInvocation()` and `codeFactory_getExternalizedInputDecl()` methods.

If the code generators set a prefix, prepend the prefix to each externalized variable name used with the `codeFactory_getConstructorInvocation()` and `codeFactory_getExternalizedInputDecl()` methods. Doing so ensures that externalized variable names in different verification points within the same scope will be unique.

The following example illustrates the use of code factory methods:

```
public int codeFactory_getNumExternalizedInputs()
   {
      int iLines = 0;

      // At least 6 lines of code, 4 for JDBC connect info, 1 for VP name and
      // 1 for SQL statement.
      iLines += 6;

      if ( getOptions() != 0 )
      {
         // If the user set any options, need to add another variable for that.
         iLines++;
      }

      return iLines;
   }


public String codeFactory_getExternalizedInputDecl( int nInput )
   {
      String sCode = "";
      String sPrefix = this.codeFactory_getPrefix();

      // Out of range request gets an empty string (still valid code...)
      if ( nInput < codeFactory_getNumExternalizedInputs() )
      {
         switch ( nInput )
         {
            case 0:
               sCode = "String s" + sPrefix + "JDBCdriver = \"" + sJDBCdriver
                           + "\";";
               break;
            case 1:
               sCode = "String s" + sPrefix + "JDBCurl = \"" + sJDBCurl + "\";";
               break;
            case 2:
               sCode = "String s" + sPrefix + "JDBCuser = \"" + sJDBCuser
                           + "\";";
               break;
            case 3:
               sCode = "String s" + sPrefix + "JDBCpassword = \""
                           + sJDBCpassword + "\";";
               break;
            case 4:
               sCode = "String s" + sPrefix + "SQL = \"" + sSQL + "\";";
               break;
            case 5:
               sCode = "String s" + sPrefix + "VPname = \"" + getVPname()
                           + "\";";
               break;
            case 6:
               sCode = "int i" + sPrefix + "Options = "
                           + Integer.toString(getOptions()) + ";";
               break;
            default:
               sCode = "";
               break;
         }
      }

      return sCode;
   }


public String codeFactory_getConstructorInvocation()
   {
      StringBuffer sCode = new StringBuffer("");
      String sPrefix = this.codeFactory_getPrefix();

      sCode.append("DatabaseVP ");
      sCode.append(sPrefix);
      sCode.append(this.getVPname());
      sCode.append(" = new DatabaseVP( \"");
      sCode.append(this.getVPname());
      sCode.append("\", s");
      sCode.append(sPrefix);
      sCode.append("SQL, s");
      sCode.append(sPrefix);
      sCode.append("JDBCuser, s");
      sCode.append(sPrefix);
      sCode.append("JDBCpassword, s");
      sCode.append(sPrefix);
      sCode.append("JDBCdriver, s");
      sCode.append(sPrefix);
      sCode.append("JDBCurl");


      if ( this.getOptions() != 0 )
      {
         sCode.append(", i");
         sCode.append(sPrefix);
         sCode.append("Options);");
      }
      else
         sCode.append(");");

      return sCode.toString();

   }
```

Step 5. Provide Serialization Services for the Metadata

Implement `readFile()` and `writeFile()` methods to serialize verification point metadata.

The metadata file is read by both the Verification Point Data Comparator class and the TestManager comparator software. Currently, the only supported metadata file format is .ini file format.

A future release of Rational QualityArchitect will support custom-built comparators in addition to the TestManager comparator. As a result, you will be able to use any metadata (and data) file format that your custom comparator supports.

When reading and writing your metadata file, store all metadata for your verification point, as well as properties for the additional `[Definition]` section in the .ini file, as shown in the following example:

```
public void writeFile(OutputStream out) throws IOException
   {
      // If there's nothing to write -- don't write anything...
      if ( sJDBCdriver == "" || sJDBCurl == "" || sSQL == "" )
         return;

      PrintWriter pwOut = new PrintWriter ( new BufferedWriter (
         new OutputStreamWriter ( out )));

      // Write out the [Definition] section
      pwOut.println("[Definition]");

      // Write the VP name
      pwOut.println("Case ID=" + this.getVPname());

      // Write the VP type
      pwOut.println("Type=Object Data");

      // Write the data test
      pwOut.println("Data Test=Contents");

      // Write the verification method
      if ( (getOptions() & COMPARE_CASEINSENSITIVE) != 0 )
         pwOut.println("Verification Method=CaseInsensitive");
      else
         pwOut.println("Verification Method=CaseSensitive");

      // Write out the DatabaseVP specific section.
      pwOut.println("");
      pwOut.println("[Database VP]");

      // Write out the JDBC connect info
      pwOut.println("JDBCdriver=" + sJDBCdriver);
      pwOut.println("JDBCurl=" + sJDBCurl);
      pwOut.println("JDBCuser=" + sJDBCuser);
      pwOut.println("JDBCpassword=" + sJDBCpassword);

      // Write out the Select statement
      pwOut.println("SQL=" + sSQL);

      // Flush the output, and close the file.
      pwOut.flush();
   }

public void readFile(InputStream in) throws IOException
   {
      try
      {
         Hashtable tblINI = CTutil.mapINIfile( in );
         if ( tblINI != null )
         {
            String sDef = "Definition";
            String sDBVP = "Database VP";

            // Read out all the entries we care about.
            String sVerMethod = CTutil.readPrivateProfileString(tblINI, sDef,
                                                  "Verification Method");
            if ( sVerMethod.equals("CaseInsensitive") )
               setOptions(getOptions()|COMPARE_CASEINSENSITIVE);

            sJDBCdriver = CTutil.readPrivateProfileString(tblINI, sDBVP,
                                                  "JDBCdriver");
            sJDBCurl = CTutil.readPrivateProfileString(tblINI, sDBVP,
                                                  "JDBCurl");
            sJDBCuser = CTutil.readPrivateProfileString(tblINI, sDBVP,
                                                  "JDBCuser");
            sJDBCpassword = CTutil.readPrivateProfileString(tblINI, sDBVP,
                                                  "JDBCpassword");
            sSQL = CTutil.readPrivateProfileString(tblINI, sDBVP, "SQL");
         }
      }
      catch ( IOException exc ) { }
      return;
   }
```

Implementing the Verification Point Data Class

Your specialized Verification Point Data class must implement the `com.rational.test.vp.VerificationPointData` interface and perform the following high-level tasks:

1. Create member variables that encapsulate the data that the verification point is comparing.
2. Implement `readFile()` and `writeFile()` methods to serializethe data to a verification point data file.
3. Implement the `getFileExtension()` method.

Step 1. Encapsulate the Data Being Compared

Create member variables that encapsulate the data that the verification point is comparing. The data encapsulated in these member variables should be exposed through public `get` and `set` methods that you implement. Doing so allows a test script to create and populate an instance of the class for use in dynamic and manual verification points.

The following example uses the public `getData()` and `setData()` methods to encapsulate the data objects being compared:

```
private String[] asColumns = null;
private Vector vData = null;

public int getNumCols()
{
   if (asColumns != null )
      return asColumns.length;
   else
      return 0;
}

public int getNumRows()
{
   if ( vData != null )
      return vData.size();
   else
      return 0;
}

public String[] getColumns()
{
   return asColumns;
}

public void setColumns( String[] asColumns )
{
   this.asColumns = asColumns;
}

public Vector getData()
{
   return vData;
}

public void setData( Vector vData )
{
   this.vData = vData;
}
```

Step 2. Serialize the Data to a Data File

Implement `readFile()` and `writeFile()` methods to serialize verification point data.

The data file is read by both the Verification Point Data Comparator class and the TestManager comparator software. Currently, the only supported data file format is .csv file format.

A future release of Rational QualityArchitect will support custom-built comparators in addition to the TestManager comparator. As a result, you will be able to use any data (and metadata) file format that your custom comparator supports.

The following example illustrates reading from and writing to a .csv file:

```
public void writeFile(OutputStream out) throws IOException
   {
      // If there's nothing to write -- don't write anything...
      if ( asColumns == null || vData == null || asColumns.length == 0 )
         return;

      PrintWriter pwOut = new PrintWriter ( new BufferedWriter (
         new OutputStreamWriter ( out )));

      // First print out a line with all the column names.
      String csvColumns = "";
      int numCols = getNumCols();
      for ( int i=0; i < numCols; i++ )
      {
         if ( i > 0 )
            csvColumns = csvColumns + "," + "\"" + asColumns[i] + "\"";
         else
            csvColumns = "\"" + asColumns[i] + "\"";
      }
      pwOut.println(csvColumns);

      // Next print out a line for each element in our vector of data.
      int numRows = getNumRows();
      for ( int i=0; i < numRows; i++ )
      {
         Object obj = vData.elementAt(i);
         if ( obj != null )
         {
            // Verify that obj is an array of strings

            String[] asData = (String[]) obj;
            if ( asData.length != numCols )
            {
               // Don't write out this row, and write an error message
               // to the log about the format of this object.

               // Log warning message here.
            }
            else
            {
               String csvRow = "";
               for ( int j=0; j < numCols; j++ )
               {
                  if ( j > 0 )
                     csvRow = csvRow + "," + "\"" + asData[j] + "\"";
                  else
                     csvRow = "\"" + asData[j] + "\"";
               }
               pwOut.println(csvRow);
            }
         }
      }

      // Flush the output.
      pwOut.flush();

   }

   public void readFile(InputStream in) throws IOException,
                     ClassNotFoundException
   {
      BufferedReader brIn = new BufferedReader (
         new InputStreamReader ( in ));

      // Read in the array of column names
      String sColumns = brIn.readLine();

      // If the file is empty, we're done.
      if ( sColumns == null || sColumns.length() == 0 )
         return;

      StringBuffer bufCSV = new StringBuffer(sColumns);
      StringBuffer bufElement = new StringBuffer("");
      int numCols = 0;
      boolean bMore = true;
      Vector vColumns = new Vector();

      while (bMore == true)
      {
         bMore = CTutil.csvGetNextElement(bufCSV, bufElement);
         String sElement = bufElement.toString();

         // Remove quotes around string if they are present.
         if ( sElement.startsWith("\"") && sElement.endsWith("\"") )
         {
            sElement = sElement.substring(1, sElement.length() - 1);
         }
         vColumns.addElement(sElement);
         numCols++;
      }

      // Turn the vector into an array of strings.
      asColumns = (String[])CTutil.toArray(vColumns, new String[1]);

      // Now read in all the data lines.
      String sData = "";
      Vector vRow = new Vector();
      vData = new Vector();

      for ( sData = brIn.readLine(); sData != null; sData = brIn.readLine() )
      {
         bufCSV = new StringBuffer(sData);
         bufElement.setLength(0);
         int numElements = 0;
         bMore = true;
         vRow.removeAllElements();

         while (bMore == true)
         {
            bMore = CTutil.csvGetNextElement(bufCSV, bufElement);
            String sElement = bufElement.toString();

            // Remove quotes around string if they are present.
            if ( sElement.startsWith("\"") && sElement.endsWith("\"") )
            {
               sElement = sElement.substring(1, sElement.length() - 1);
            }
            vRow.addElement(sElement);
            numElements++;
         }

         if ( numElements == numCols )
         {
            vData.addElement(CTutil.toArray(vRow, new String[1]));
         }
         else
         {
            // Handle the exception.
         }
      }
   }
```

Step 3. Provide the Extension for the Data File

Call `getFileExtension()` to provide the extension of the data file to the test script.

In this release of QualityArchitect, this method always returns csv. In a future release, the method will return the file extension used by whatever data file format (for example, .csv, .dat, .xml) that you select for the data in your Verification Point Data class.

The verification point framework creates the unique file name and data file passed to the `writeFile()` and `readFile()` methods. The `getFileExtension()` method tells the framework what file extension to use, as shown in the following example:

```
public String getFileExtension()
{
   return "csv";
}
```

Implementing the Verification Point Data Comparator Class

Your specialized Verification Point Data Comparator class must implement the `com.rational.test.vp.VerificationPointDataComparator` interface.

The only method in this interface is `compare()`. This method compares an expected data object with an actual data object (both of type `VerificationPointData`) and determines whether the test passes or fails.

The following example illustrates a comparison of two data objects:

```
public boolean compare( VerificationPointData vpdExpected,
                     VerificationPointData vpdActual,
                     Object objOptions,
                     StringBuffer sFailureDescription )
   {
      boolean bIdentical = true;
      StringBuffer bufActual = new StringBuffer();
      StringBuffer bufExpected = new StringBuffer();
      StringBuffer bufFailIndex = new StringBuffer();
      Integer iOptions;

      if ( objOptions != null )
         iOptions = (Integer) objOptions;
      else
         iOptions = new Integer(0);

      boolean bCaseInsensitive = (iOptions.intValue() &
                     VerificationPoint.COMPARE_CASEINSENSITIVE) != 0;

      DatabaseVPData expected = (DatabaseVPData) vpdExpected;
      DatabaseVPData actual = (DatabaseVPData) vpdActual;

      if ( expected.getNumCols() != actual.getNumCols() )
      {
         String sText;
         if ( expected.getNumCols() == 0 || actual.getNumCols() == 0 )
            sText = "No column titles";
         else
            sText = "Differing number of columns";

         sFailureDescription.insert(0, sText);
         sFailureDescription.setLength(sText.length());
         return false;
      }
      if ( expected.getNumRows() != actual.getNumRows() )
      {
         String sText = "Differing number of rows";
         sFailureDescription.insert(0, sText);
         sFailureDescription.setLength(sText.length());
         return false;
      }
      if ( compareStringArray( expected.getColumns(), actual.getColumns(),
                               bCaseInsensitive, bufExpected, bufActual,
                               bufFailIndex) == false )
      {
         String sText = "Column title[" + bufFailIndex.toString() +
                        "]: expected[";
         sText += bufExpected.toString() + "], actual[" +
                        bufActual.toString() + "].";
         sFailureDescription.insert(0, sText);
         sFailureDescription.setLength(sText.length());
         return false;
      }

      // Walk the vectors of data and compare each row.
      int numRows = expected.getNumRows();
      int numCols = expected.getNumCols();
      Vector vExpected = expected.getData();
      Vector vActual = actual.getData();
      String[] asExpected;
      String[] asActual;

      for ( int i=0; i < numRows; i++ )
      {
         Object obj = vExpected.elementAt(i);
         asExpected = (String[]) obj;

         obj = vActual.elementAt(i);
         asActual = (String[]) obj;

         if ( compareStringArray( asExpected, asActual, bCaseInsensitive,
               bufExpected, bufActual, bufFailIndex ) == false )
         {
            // Row + 2 -> 1 for the column titles (which show up as a row)
            // and one for 0 index into vector vs. 1 index in grid comparator.
            String sText = "Difference found in row[" + Integer.toString(i+2);
            sText += "], column[" + bufFailIndex.toString() + "].";
            sFailureDescription.insert(0, sText);
            sFailureDescription.setLength(sText.length());
            return false;
         }
      }

      return true;
   }
```

```
private boolean compareStringArray( String[] asX, String[] asY,
            boolean bCaseInsensitive, StringBuffer bufFailX,
            StringBuffer bufFailY, StringBuffer bufFailIndex )
{
   if ( asX.length != asY.length )
      return false;

   boolean bDifferent;

   for ( int i=0; i < asX.length; i++ )
   {
      if ( bCaseInsensitive )
         bDifferent = !asX[i].equalsIgnoreCase(asY[i]);
      else
         bDifferent = !asX[i].equals(asY[i]);

      if ( bDifferent )
      {
         bufFailIndex.insert(0, Integer.toString(i+1));
         bufFailIndex.setLength(Integer.toString(i).length());
         bufFailX.insert(0, asX[i]);
         bufFailX.setLength(asX[i].length());
         bufFailY.insert(0, asY[i]);
         bufFailY.setLength(asY[i].length());
         return false;
      }
   }
   return true;
}
```

Implementing the Verification Point Data Provider Class

Your specialized Verification Point Data Provider class must implement the `com.rational.test.vp.VerificationPointDataProvider` interface.

The only method in this interface is `captureData()`. This method uses the metadata in a `VerificationPoint` object to construct and populate a `VerificationPointData` object.

The following example illustrates an implementation of the `captureData()` method:

```
public VerificationPointData captureData( java.lang.Object theObject,
                                          VerificationPoint VP )
   {
      DatabaseVP theVP = (DatabaseVP) VP;
      String sSQL = theVP.getSQL();
      String sJDBCuser = theVP.getJDBCuser();
      String sJDBCpassword = theVP.getJDBCpassword();
      String sJDBCdriver = theVP.getJDBCdriver();
      String sJDBCurl = theVP.getJDBCurl();
      int iOptions = theVP.getOptions();
```

```
Connection con = theVP.getCon();
Statement stmt = theVP.getStmt();

DatabaseVPData vpsData = null;

// Capture the data!

if ( con == null || stmt == null )
{
   // Create a JDBC connection and statement
   try {
      Class.forName(sJDBCdriver);

   }
   catch(ClassNotFoundException e) {
      theVP.sFailureDescription =
               "Database VP Error: Unable to load driver \""
               + sJDBCdriver + "\"";
      theVP.bIsValid = false;
      return vpsData;
   }
   try {
      con = DriverManager.getConnection(sJDBCurl, sJDBCuser,
                                        sJDBCpassword);
   }
   catch(SQLException ex) {
      theVP.sFailureDescription =
         "Database VP Error: Unable to Connect, UID = "
         + sJDBCuser + ", PWD = " + sJDBCpassword + ", URL = "
         + sJDBCurl + ", Error = " + ex.getMessage();
      theVP.bIsValid = false;
      return vpsData;
   }
   try {
      stmt = con.createStatement();
   }
   catch(SQLException ex) {
      theVP.sFailureDescription =
               "Database VP Error: Unable to create Statement: "
               + ex.getMessage();
      theVP.bIsValid = false;
      return vpsData;
   }
}

// Execute the query.
try {
   ResultSet rs = stmt.executeQuery(sSQL);
   ResultSetMetaData rsmd = rs.getMetaData();

   vpsData = new DatabaseVPData();
   int numColumns = rsmd.getColumnCount();
   String[] asColumns = new String[numColumns];
```

```
// Build a String array of the Column Names
if ( (iOptions & DatabaseVP.OPTION_TRIM) != 0 )
{
   for (int i=0; i < numColumns; i++)
   {
      asColumns[i] = rsmd.getColumnName(i+1).trim();
   }
}
else
{
   for (int i=0; i < numColumns; i++)
   {
      asColumns[i] = rsmd.getColumnName(i+1);
   }
}

// Put the column data into the VPdata object
vpsData.setColumns(asColumns);

// Build a Vector of the data elements
Vector vData = new Vector();
int numRows = 0;
try {
   while( rs.next() )
   {
      String[] asData = new String[numColumns];
      if ( (iOptions & DatabaseVP.OPTION_TRIM) != 0 )
      {
         for (int j=0; j < numColumns; j++)
         {
            asData[j] = rs.getString(j+1).trim();
         }
      }
      else
      {
         for (int j=0; j < numColumns; j++)
         {
            asData[j] = rs.getString(j+1);
         }
      }

      // Put the array of strings into the vector at this row's
      index.
      vData.addElement((Object) asData);
      numRows++;
   }
}
catch(SQLException ex) {
   theVP.sFailureDescription =
         "Database VP Error: Unable to walk ResultSet.  "
         + "Error = " + ex.getMessage();
   theVP.bIsValid = false;
   return null;
```

```
            }
            vpsData.setData(vData);
         }
         catch(SQLException ex) {
            theVP.sFailureDescription =
                     "Database VP Error: Unable to execute Query \""
                     + sSQL + "\", Error = " + ex.getMessage();
            theVP.bIsValid = false;
            return vpsData;
         }

         return vpsData;
      }
```

Implementing the Verification Point Data Renderer Class

Your specialized Verification Point Data Renderer class must implement the `com.rational.test.vp.VerificationPointDataRenderer` interface.

The only method in this interface is `displayAndValidateData()`. This method:

- Displays the data in a `VerificationPointData` object
- Allows the user to accept or reject that data as being correct.

The verification point framework invokes `displayAndValidateData()` when both of the following conditions apply:

- You execute a static verification point for the first time (that is, when the baseline data is first captured).
- The test designer has specified the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method of the specialized `VerificationPoint` class.

When both of these conditions exist, the framework captures the baseline data object and then invokes `displayAndValidateData()` to display the baseline data. The tester accepts or rejects the data:

- If the tester accepts the data as being correct, the framework stores the data as the baseline for the static verification point.
- If the tester rejects the data, the framework does not store the baseline data for the verification point. The process repeats the next time you execute the verification point.

In the following example, `displayAndValidateData()` presents the baseline data object `vpdData` to the tester for verification:

```
public boolean displayAndValidateData( VerificationPointData vpdData )
{
   // Pop up some UI which displays the vpdData object and prompts the
   // user to accept or reject.

   if ( bAccepted )
      return true;
   else
      return false;
}
```

Integrating a Verification Point with QualityArchitect

Once you have implemented a verification point, integrate the verification point into the QualityArchitect environment. After you do so, testers will be able to insert your verification point into a test script when they generate a test script from a Rational Rose model or when they record a test script with the Session Recorder.

To integrate your verification point with QualityArchitect, perform both of these tasks:

- Register the verification point in the rqalocvp.ini file. This file lists custom verification point types in the section JAVA VP in the following format:

  VpType = PackageSpecificationName

  The following is an example of how the database verification point, which is part of the `com.rational.test.vp` package provided with QualityArchitect, would be registered in the .ini file:

  ```
  [Java VP]
  DatabaseVP = com.rational.test.vp.DatabaseVP
  ```

  The rqalocvp.ini file is located in the Rational datastore in the folder DefaultTestScriptDataStore.

- Add the .jar file containing your new verification point classes to the CLASSPATH. For more information about CLASSPATH settings, see *Running Test Scripts* on page 8.

Verification Point Framework Reference 8

About the Verification Point Framework

The verification point *framework* is the underlying "machinery" that executes and manages a verification point. The framework serves two purposes:

- It provides the base class and interfaces that a verification point implementer uses to create a new verification point.
- In a fully implemented verification point, it performs much of the functionality of a verification point "under the covers," shielding the test designer and the verification point implementer from having to code this functionality explicitly.

Requirements for Using the Verification Point Framework

Use of the verification point framework requires Rational QualityArchitect.

In addition, the CLASSPATH must reference a number of .jar files. For a list of the required .jar files, see *Running Test Scripts* on page 8.

Components of the Verification Point Framework

The verification point framework contains the following class and interfaces:

- *VerificationPoint Class* on page 182
- *VerificationPointComparator Interface* on page 199
- *VerificationPointData Interface* on page 201
- *VerificationPointDataProvider Interface* on page 203
- *VerificationPointDataRenderer Interface* on page 205

This class and the interfaces are included in the package `com.rational.test.vp`.

VerificationPoint Class

This class contains the verification point's *metadata* — that is, the information that determines the data to capture for this verification point. Examples of verification point metadata include the list of properties for a user-defined object properties verification point, or connection information and SELECT statements for the JDBC database verification point that is included in the `com.rational.test.vp` package.

Don't confuse metadata with the data being verified. The data being verified is encapsulated by an implementation of the interface `VerificationPointData`.

A verification point's metadata can be defined in either of these ways:

- Explicitly, through the constructor or through user-defined `set...` methods in your specialized `VerificationPoint` class.
- Implicitly, through metadata retrieved from the datastore.

  If the metadata has not been explicitly specified and no metadata exists for this verification point in the datastore, the framework calls the `defineVPcallback()` method in your specialized `VerificationPoint` class. Your implementation of this method should provide some means of retrieving the verification point's metadata— typically through some UI that prompts the tester for the information. When the metadata is retrieved, the framework stores it in the datastore.

For more information about specifying metadata, see *Step 1. Specify the Metadata for the Verification Point* on page 127.

This class must also implement its own serialization. By requiring your specific verification point implementations to perform their own serialization, you can support all file formats (such as INI, XML, and standard Java serialization).

Note: The current release only supports the vpm and .ini formats.

This `abstract` class defines the metadata for and partially implements the behavior of a verification point. Because the `VerificationPoint` class is abstract, it cannot be instantiated. Rather, all verification point classes, including the classes you create, extend from this class, implementing the abstract methods necessary to specialize themselves, and inheriting the rest of their behavior from this class.

As the verification point implementer, you must implement all abstract methods.

Overview

```
public abstract class VerificationPoint
extends java.lang.Object

java.lang.Object
  |
  +--com.rational.test.vp.VerificationPoint

Known subclass:
DatabaseVP
```

Applicability

Commonly used with Rational QualityArchitect.

This class requires Rational QualityArchitect.

Summary

This class contains the following fields:

Field	**Description**
`bIsDefined`	`protected boolean`. If `true`, indicates that the verification point's metadata is fully specified. If `false` when a `performTest()` method is invoked, the framework will call the `defineVPcallback()` method on behalf of the test script in an attempt to get a full set of verification point metadata from the tester. Note that this field applies to the verification point *metadata*, not to the data itself that is captured in accordance with the metadata.
`bIsValid`	`protected boolean`. If `true`, indicates that the verification point was correctly instantiated, successfully captured, and is in a valid state; otherwise, `false`.
`COMPARE_CASEINSENSITIVE`	`static int`. Specifies that the verification should be case insensitive.
`COMPARE_CASESENSITIVE`	`static int`. Specifies that the verification should be case sensitive (default).
`OPTION_EXPECT_FAILURE`	Specifies that the Verification Point's expected result is failure. If the comparison fails and this option is set, the verification point succeeds.

Field	**Description**
`OPTION_USER_ ACKNOWLEDGE_BASELINE`	`static int`. 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.
`sFailureDescription`	`protected java.lang.String`. Specifies the reason for a failure.
`VERIFICATION_ERROR`	`static int`. Indicates that an error occurred, and the verification point was not performed.
`VERIFICATION_FAILED`	`static int`. Indicates that the verification point was performed, and the comparison failed.
`VERIFICATION_NO_RESULT`	`static int`. Indicates that the static verification point was run for the first time, and a baseline (expected) data object was successfully captured.
`VERIFICATION_SUCCEEDED`	`static int`. Indicates that the verification point was performed, and the comparison passed.

This class contains the following methods:

Method	Description
`codeFactory_ getConstructorInvocation()`	Returns a parameterized constructor call.
`codeFactory_ getExternalizedInputDecl()`	Returns a variable declaration.
`codeFactory_ getNumExternalizedInputs()`	Returns the number of responses (inputs) that a tester provided when defining verification point metadata interactively through a UI.
`codeFactory_getPrefix()`	Retrieves the user-defined prefix that is currently available to prepend to a variable name to make the name unique.
`codeFactory_setPrefix()`	Specifies a user-defined prefix to prepend to the current set of variable names. The names are created and declared by the `codeFactory_getExternalizedInputDecl()` method.
`defineVPcallback()`	Provides a way to capture the metadata for the verification point — typically, by presenting the tester with a UI device, such as the Query Builder tool provided with Rational QualityArchitect (for use with the database verification point).
`getIsDefined()`	Retrieves the value of the `bIsDefined` field.
`getOptions()`	Retrieves the options associated with the current verification point.
`getVPname()`	Retrieves the name of the current verification point.
`performTest()`	Performs a static, dynamic, or manual verification point, depending upon the parameters that are passed to it.
`readFile()`	Deserializes a verification point object from the specified InputStream.
`setIsDefined()`	Sets a value for the `bIsDefined` field.
`setOptions()`	Sets the options for the current verification point.
`setVPname()`	Sets the name of the current verification point.
`writeFile()`	Serializes the verification point object to the specified OutputStream.

Methods Inherited from Class java.lang.Object
`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Note: For more information about these code factory methods, see *Step 4. Implement the Code Factory Methods to Generate Code* on page 165.

Constructor

This constructor stores the name of the verification point and the classes that provide serialization and comparison services for the verification point.

Syntax

```
public VerificationPoint(java.lang.String sVPname,
   java.lang.Class cVPdataClass, java.lang.Class
   cVPdataProviderClass, java.lang.Class cVPdataRendererClass,
   java.lang.Class cVPcompClass)
```

Element	**Description**
sVPname	A user-defined name for the verification point (40 characters maximum).
cVPdataClass	The class responsible for serialization of one set of the verification point's data.
cVPdataProviderClass	The class responsible for capturing the verification point's data and populating a `VerificationPointData` object with the data.
cVPdataRendererClass	The class responsible for visually rendering the data stored in an object of this verification point's `VerificationPointData` class.
cVPcompClass	The class responsible for comparing two sets of this verification point's data.

Comments

The classes passed in the `cVPdataClass`, `cVPdataProviderClass`, `cVPdataRendererClass`, and `cVPcompClass` parameters are passed to the constructor to allow late-binding to the methods in the classes.

A test script can never call this constructor because the `VerificationPoint` class is abstract. A specific verification point implementation (such as the pre-defined database verification point or any custom verification points that you implement) extends the `VerificationPoint` class and invokes this constructor from within its own constructor using the `super` keyword — for example:

```
public MyVerificationPoint( String sVPname )
   {
   super(sVPname, MyVPData.class, MyVPDataProvider.class,
         MyVPDataRenderer.class, MyVPComparator.class);
   . . .
   }
```

VerificationPoint.codeFactory_getConstructorInvocation()

Returns a parameterized constructor call.

Syntax

```
public abstract java.lang.String
    codeFactory_getConstructorInvocation()
```

Comments

The test script never calls this method. This method is called during the following Rational code generation operations:

- Rational QualityArchitect test script recording
- Rational Rose scenario test generation

This method returns a parameterized constructor call. This call plus the variables declared by `codeFactory_getExternalizedInputDecl()` allow these code generators to create a fully specified verification point in the generated test script code.

For more information, see *Step 4. Implement the Code Factory Methods to Generate Code* on page 165.

VerificationPoint.codeFactory_getExternalizedInputDecl()

Returns a variable declaration.

Syntax

```
public abstract java.lang.String
   codeFactory_getExternalizedInputDecl(int nInput)
```

Element	Description
nInput	A number that indicates the current variable to declare. The number should be initialized to 0 and incremented by 1 in a loop.

Return Value

A line of code that declares the specified variable.

Comments

The test script never calls this method. This method is called during the following Rational code generation operations:

- Rational QualityArchitect test script recording
- Rational Rose scenario test generation

The Rational code generators (the Rational QualityArchitect Session Recorder and the Rational Rose scenario test generator) call this method in a loop that iterates as many times as there are variables to declare (that is, the number returned from `codeFactory_getNumExternalizedInputs()`).

The code returned by `codeFactory_getConstructorInvocation()` uses the variables declared with `codeFactory_getExternalizedInputDecl()`.

VerificationPoint.codeFactory_getNumExternalizedInputs()

Returns the number of responses (inputs) that a tester provided when defining verification point metadata interactively through a UI. The UI was presented to the tester through the `defineVPcallback()` method.

Syntax

```
public abstract int codeFactory_getNumExternalizedInputs()
```

Return Value

The number of inputs that require variable declarations.

Comments

The test script never calls this method. This method is called during the following Rational code generation operations:

- Rational QualityArchitect test script recording.
- Rational Rose scenario test generation.

VerificationPoint.codeFactory_getPrefix()

Retrieves the user-defined prefix that is currently available to prepend to a variable name to make the name unique.

Syntax

```
public java.lang.String codeFactory_getPrefix()
```

Return Value

A prefix for a variable name.

Comments

Call this method only if you are implementing a new verification point.

Use the prefix whenever you are constructing a set of variable names for use with the `codeFactory_getConstructorInvocation()` and `codeFactory_getExternalizedInputDecl()` methods.

VerificationPoint.codeFactory_setPrefix()

Specifies a user-defined prefix to prepend to the current set of variable names created and declared by the `codeFactory_getExternalizedInputDecl()` method.

Syntax

```
public void codeFactory_setPrefix(java.lang.String
   sSelfDescribePrefix)
```

Element	Description
sSelfDescribedPrefix	The prefix to prepend to the variable names.

Comments

The variable-name prefix ensures that variable names are unique when the Rational code generators (the Rational QualityArchitect Session Recorder and the Rational Rose scenario test generator) insert more than one verification point into a given scope.

Rational QualityArchitect code generators call this method.

VerificationPoint.defineVPcallback()

Provides a way to capture the metadata for the verification point — typically, by presenting the tester with a UI device, such as the Query Builder tool provided with Rational QualityArchitect (for use with the database verification point).

Syntax

```
public abstract boolean defineVPcallback()
```

Return Value

`true` if the verification point metadata was captured; otherwise, `false`. If the metadata was not captured, the verification point will be in an invalid state, and it will log an error if its `performTest()` method is called.

Comments

The verification point framework automatically invokes this method if the verification point is not fully defined when you invoke the `performTest()` method.

When you invoke `defineVPcallback()`, it should do the following (presumably through a UI):

1 Capture any information necessary for fully defining the metadata for the verification point

2 Populate the verification point's attributes with the captured metadata.

For example, the `defineVPcallback()` method included with the database verification point provided with Rational QualityArchitect invokes the Query Builder software. Query Builder captures JDBC connection information and a SQL statement, and then populates the database verification point object with the captured metadata, resulting in a fully defined verification point.

This method applies to the verification point *metadata*, not to the data itself that is captured in accordance with the metadata. The specialized Verification Point Data Provider class uses the metadata to determine which data to capture.

If Rational QualityArchitect Session Recorder is recording the verification point, or if a Rational Rose model is generating it, this method will be invoked at script generation time. The resulting verification point metadata will automatically be provided to the test script. As a result, the `defineVPcallback()` method will not be invoked at script playback time.

Implement this method only if you are implementing a new verification point.

VerificationPoint.getIsDefined()

Retrieves the value of the `bIsDefined` field.

Syntax

```
public boolean getIsDefined()
```

Return Value

If `true`, the verification point's metadata was fully specified in the constructor call. If `false`, the metadata was not fully specified.

Comments

If the verification point metadata is not defined when `performTest()` is called, the framework will call the `defineVPcallback()` method on behalf of the test script in an attempt to get a complete set of verification point metadata from the tester.

See Also

`getIsDefined()` on page 191

VerificationPoint.getOptions()

Retrieves the options associated with the current verification point.

Syntax

```
public int getOptions()
```

Return Value

The options associated with the current verification point.

See Also

`setOptions()` on page 198

VerificationPoint.getVPname()

Retrieves the name of the current verification point.

Syntax

```
public java.lang.String getVPname()
```

Return Value

The name of the current verification point.

See Also

`setVPname()` on page 198

VerificationPoint.performTest()

Performs a static verification point.

Syntax

```
public int performTest(java.lang.Object objTarget)
```

Element	Description
`objTarget`	The object-under-test. If the verification point operates on an object that is not directly accessible (for example, a remote object or a database), the verification point object must contain the information needed to find the object-under-test, and the value of *`objTarget`* is ignored. This parameter is passed to `captureData()` as its first parameter.

Comments

In this implementation, `performTest()` performs and logs a regression-style verification. It does so by checking the datastore for an expected (baseline) data object, and then comparing the expected data object to the actual data object that is captured in this call.

If there is no expected data object in the datastore, the framework creates one and the method returns a `VERIFICATION_NO_RESULT` for this run of the verification point.

However, if there is no expected data, but the test script specifies the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method, the framework first invokes an implementer-defined UI that prompts the tester to verify that the captured data is correct:

- If the tester accepts the displayed data as being correct, the framework stores the data object in the datastore as the expected data for subsequent tests, and the method returns `VERIFICATION _SUCCEEDED`.
- If the tester rejects the displayed data, the framework logs an error, and verification point execution ends. The framework does not store an expected data object.

Return Value

This method returns one of the following values:

Value	Description
`VERIFICATION_SUCCEEDED`	The verification point was performed, and the comparison passed.
`VERIFICATION_FAILED`	The verification point was performed, and the comparison failed.
`VERIFICATION_NO_RESULT`	The static verification point was run for the first time, and a baseline (expected) data object was successfully captured.
`VERIFICATION_ERROR`	An error occurred, and the verification point was not performed.

VerificationPoint.performTest()

Performs a dynamic verification point.

Syntax

```
public int performTest(java.lang.Object objTarget,
   VerificationPointData vpsExpected)
```

Element	Description
objTarget	The object-under-test. If the verification point operates on an object that is not directly accessible (for example, a remote object or a database), the verification point object must contain the information needed to find the object-under-test, and the value of *objTarget* is ignored. This parameter is passed to `captureData()` as its first parameter.
vpsExpected	An expected data object. The test script can construct the expected data object, or it can deserialize the expected data object from a file that is not managed by the datastore.

Return Value

This method returns one of the following values:

Value	Description
`VERIFICATION_SUCCEEDED`	The verification point was performed, and the comparison passed.
`VERIFICATION_ERROR`	An error occurred, and the verification point was not performed.
`VERIFICATION_FAILED`	The verification point was performed, and the comparison failed.

Comments

In this implementation, `performTest()` captures an actual data object from the component-under-test, compares the actual data object to the expected data object that was passed to the call, and logs the results of the comparison.

VerificationPoint.performTest()

Performs a manual verification point.

Syntax

```
public int performTest(java.lang.Object objTarget,
   VerificationPointData vpsExpected, VerificationPointData
   vpsActual)
```

Element	Description
objTarget	The object-under-test. If the verification point operates on an object that is not directly accessible (for example, a remote object or a database), the verification point object must contain the information needed to find the object-under-test, and the value of *objTarget* is ignored. This parameter is passed to `captureData()` as its first parameter.
vpsExpected	An expected data object. The test script can construct the expected data object, or it can deserialize the expected data object from a file that is not managed by the datastore.

Element	Description
vpsActual	The actual data object. The code in the test script captured or constructed this object.

Return Value

This method returns one of the following values:

Value	Description
VERIFICATION_SUCCEEDED	The verification point was performed, and the comparison passed.
VERIFICATION_ERROR	An error occurred, and the verification point was not performed.
VERIFICATION_FAILED	The verification point was performed, and the comparison failed.

Comments

In this implementation, `performTest()` specifies both the expected data object and the actual data object. This allows a test script to capture or construct the actual data object, rather than relying on the Verification Point Data Provider class to create the actual data object.

This call simply compares the actual and expected data objects that are passed to it and logs the results of the comparison.

VerificationPoint.readFile()

Deserializes a verification point object from the specified InputStream.

Syntax

```
public abstract void readFile(java.io.InputStream in)
```

Element	Description
in	The InputStream from which the object is read.

Exceptions

This method throws the following exception:

- `java.io.IOException`. An error has occurred in attempting to read from the InputStream.

VerificationPoint.setIsDefined()

Assigns a value to the `bIsDefined` field.

Syntax

```
public void setIsDefined(boolean bIsDefined)
```

Element	**Description**
`bIsDefined`	If `true`, the verification point's metadata is fully specified. If `false`, the metadata is not fully specified.

Comments

If the verification point metadata is not defined when `performTest()` is called, the framework will call the `defineVPcallback()` method on behalf of the test script in an attempt to get a complete set of verification point metadata from the tester.

See Also

`getIsDefined()` on page 191

VerificationPoint.setOptions()

Sets the options for the current verification point.

Syntax

```
public void setOptions(int iOptions)
```

Element	Description

iOptions	One or more options to assign to the verification point. Options can be pre-defined, as in the following:
	`COMPARE_CASESENSITIVE`
	`COMPARE_CASEINSENSITIVE`
	`OPTION_EXPECT_FAILURE`
	Options can also be any user-defined options.

See Also

`getOptions()`

VerificationPoint.setVPname()

Assigns a name to the current verification point.

Syntax

```
public void setVPname(java.lang.String sVPname)
```

Element	Description
sVPname	The name to assign to the current verification point (40 characters maximum).

See Also

`getVPname()`

VerificationPoint.writeFile()

Serializes the verification point object to the specified OutputStream.

Syntax

```
public abstract void writeFile(java.io.OutputStream out)
```

Element	Description
out	The OutputStream to which the object is written.

Exceptions

This method throws the following exception:

- `java.io.IOException`. An error has occurred in attempting to write to the OutputStream.

Comments

Metafile format is used so that the Rational comparators can read the file. For information, see *Step 5. Provide Serialization Services for the Metadata* on page 168.

VerificationPointComparator Interface

For a class implementing this interface, the interface provides a method to compare two `VerificationPointData` objects to determine if the comparison succeeds or fails. The comparison can test for equality between the expected and actual data, or it can test for some other condition (for example, that the actual data falls within a given range).

This class is passed into the constructor of the abstract `VerificationPoint` class and is used when that verification point needs to perform its comparison.

Overview

```
public interface VerificationPointComparator

Known implementing class:
DatabaseVPComparator
```

Applicability

Commonly used with Rational QualityArchitect.

This interface requires Rational QualityArchitect.

VerificationPointComparator.compare()

This method does the following:

- Compares an expected data object and an actual data object, both of type `VerificationPointData`
- Determines whether the test succeeds or fails.

```
public boolean compare(VerificationPointData vpsExpected,
   VerificationPointData vpsActual, java.lang.Object
   objOptions, java.lang.StringBuffer sFailureDescription)
```

Element	**Description**
vpsExpected	The expected data object.
vpsActual	The actual data object.
objOptions	Options that are passed from the Verification Point class to qualify the comparison. Options can include the pre-defined `COMPARE_CASESENSITIVE` and `COMPARE_CASEINSENSITIVE` options, plus any user-defined options.
sFailureDescription	An output parameter that contains the differences between the expected and actual data objects in a failed verification point. The failure description is written to the log.

Return Value

A boolean value indicating whether the test passed or failed.

VerificationPointData Interface

A class implementing this interface encapsulates and serializes a single snapshot of either expected or actual data. It can be populated through the `captureData` method of a Verification Point Data Provider class, or it can be populated manually in the test script — for example, by literal values or by values from a datapool.

Each implementation of the `VerificationPointData` interface must provide its own serialization methods in order to support all possible file formats. Use the `readFile()` and `writeFile()` methods to implement serialization for the encapsulated data.

Note: For the current Rational QualityArchitect release, Verification Point Data classes must serialize to a .CSV file format. This restriction will be removed in a future release of Rational QualityArchitect.

In addition to implementing the methods defined by this interface, all Verification Point Data classes should create member variables that encapsulate the data being compared by the verification point. The data encapsulated in these member variables should be exposed through public `get...` and `set...` methods that you implement, thereby allowing a test script to create and populate an instance of the class for use in dynamic and manual verification points.

Overview

```
public interface VerificationPointData

Known implementing class:
DatabaseVPData
```

Applicability

Commonly used with Rational QualityArchitect.

This interface requires Rational QualityArchitect.

VerificationPointData.getFileExtension()

Returns the extension of the file used to store the data object.

Syntax

```
public java.lang.String getFileExtension()
```

Return Value

The extension of the file used to store the data object.

Comments

The verification point framework uses the file extension to determine the format to use when it serializes files (for example, a .CSV extension indicates a comma-separated-value text file).

The current release only supports the .CSV file formatt. Future releases will support other formats.

VerificationPointData.readFile()

Reads the expected or actual data object from the specified InputStream.

Syntax

```
public void readFile(java.io.InputStream in)
```

Element	**Description**
in	The InputStream from which the object is read.

Exceptions

This method throws the following exception:

- `IOException`. An input/output error has occurred.

VerificationPointData.writeFile()

Writes the expected or actual data object to the specified OutputStream.

Syntax

```
public void writeFile(java.io.OutputStream out)
```

Element	**Description**
out	The OutputStream to which the object is written.

Exceptions

This method throws the following exception:

- `IOException`. An input/output error has occurred.

VerificationPointDataProvider Interface

An implementation of this class creates a Verification Point Data object based on the verification point metadata in the specialized Verification Point object.

A class implementing this interface is a pluggable link between a Verification Point class, which defines a verification point's metadata, and a Verification Point Data class, which encapsulates and serializes the data for a verification point.

When you implement a Verification Point Data class from this interface, you implement the `captureData()` method for populating a Verification Point Data object for a given Verification Point object. The Verification Point Data Provider class knows about the structure of both the Verification Point Data class, which it is building, and the Verification Point class, which specifies the data to capture.

This is an important abstraction for general types of verification points (such as object data or object properties) where many different objects may provide access to the same type of data.

An implementation of this interface can be plugged into an existing verification point implementation to provide verification point data from a new verification point data source.

You can use an implementation of this interface with static verification points (for building expected and actual data objects) and with dynamic verification points (for building actual data objects only).

Overview

```
public interface VerificationPointDataProvider

Known implementing class:
DatabaseVPDataProvider
```

Applicability

Commonly used with Rational QualityArchitect.

This interface requires Rational QualityArchitect.

VerificationPointDataProvider.captureData()

Builds a `VerificationPointData` object.

Syntax

```
public VerificationPointData captureData(java.lang.Object
   theObject, VerificationPoint theVP)
```

Element	Description
theObject	The object-under-test. The contents of this parameter are provided by the first parameter of the `performTest()` method.
theVP	The Verification Point object that contains the verification point's metadata.

Return Value

This method returns an instance of the specialized `VerificationPointData` class populated with the captured data.

Comments

This method captures data according to the metadata in the `VerificationPoint` class. The verification point framework can use the returned `VerificationPointData` object as either an expected or an actual data object.

VerificationPointDataRenderer Interface

Using a class implementing this interface, you can display the data stored in the Verification Point Data class. This enables the tester to interactively accept or reject that data as the expected (baseline) data for a static verification point.

To enable the data display, the test script sets the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method of the specialized Verification Point class.

Overview

```
public interface VerificationPointDataRenderer

All Known implementing class:
DatabaseVPDataRenderer
```

Applicability

Commonly used with Rational QualityArchitect.

This interface requires Rational QualityArchitect.

VerificationPointDataRenderer.displayAndValidateData()

Presents the tester with a visual representation of the data object as it exists before expected (baseline) data is stored for this static verification point.

Syntax

```
public boolean displayAndValidateData(VerificationPointData
   vpdData)
```

Element	**Description**
vpdData	The data to present to the tester for confirmation.

Return Value

`true` if the tester accepts the displayed data, `false` if the tester rejects the data.

Comments

The verification point framework invokes this method is invoked by the verification point framework when the following conditions exist:

- The test script sets the `OPTION_USER_ACKNOWLEDGE_BASELINE` option in the `setOptions()` method of the Verification Point class.
- No expected data object exists in the datastore when the test script calls the `performTest()` method of the Verification Point class for a static verification point.

When you invoke the method, it presents the tester with a visual representation of the data, and allows the tester to accept or reject the data:

- If the tester accepts the data, the verification point passes, and the framework adds the data to the datastore as the expected data for subsequent test runs.
- If the tester rejects the data, the framework logs the failure, and does not store the expected data for the verification point. The next time the tester runs the script, the tester is again prompted to accept the data.

Configuring Datapools, Synchronization Points, and Shared Variables

A

About Script Configuration

During execution of a test script that uses datapools, synchronization points, or shared variables, TestManager must be able to access and apply values at different points in the script, for different virtual testers. In this manual, the procedures that allow TestManager to do this efficiently are referred to as *configuration*. This appendix describes the configuration procedures.

Datapool Configuration

A test script that uses a datapool must include, somewhere in its body, a block of code such as the following:

```
public static class DatapoolConfig extends DatapoolInfo {
        public DatapoolConfig() {
            setDatapoolName(java.lang.String name);
            setDatapoolAccessFlags(int accessFlags);
        }
    }
```

The *`name`* argument of `setDatapoolName()` — a method of `DatapoolInfo` — is the same as the *`name`* argument of `TSSDatapool.open()`, and should contain the same value. Thus, if with `open()` you specify a datapool named `custdata`, specify `custdata` with `setDatapoolName()` also.

The *`accessFlags`* argument of `setDatapoolAccessFlags()` — also a method of `DatapoolInfo` — accepts the same values as argument *`accessFlags`* of the datapool `open()` method. If `open()` specifies no accesss flags, then the values you specify with `setDatapoolAccessFlags()` apply. If `open()` specifies access flags, they are ORed to flags specified with `setDatapoolAccessFlags()`. If access flags specified with `open()` contradict those specified with `setDatapoolAccessFlags()`, a `TSS_INVALID` error occurs.

The following is an example of a Java program that opens and configures a datapool named `squaredp`. Relevant lines apear in bold.

```
/*
* SquareClientTM demonstrates an EJB client that can be executed
```

```
 * from Rational Suite TestStudio using TestManager.
*/

// EJB itself
import com.rational.square.Square;
import com.rational.square.SquareHome;

// Misc
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.ListResourceBundle;
import java.rmi.RemoteException;

// JNDI-related
import javax.naming.Context;
import javax.naming.InitialContext;

// TestManager
import com.rational.test.tss.*;


// Java test scripts must extend the TestScript interface.

public class SquareClientTM_datapool extends
   com.rational.test.tss.TestScript {

    public void testMain(String[] args) {
        try {

            // Create EJB
            TSSMeasure.commandStart("home001", "getHome",
                                    MST_XCLNTCONN);
            Square square = getHome().create();
            TSSMeasure.commandEnd((short)TSS_CMD_STAT_PASS);

            // Call Square method
            long answer = 0;
            TSSDatapool dp = new TSSDatapool();
            dp.open("squaredp");
            boolean bret = dp.fetch();
            int dpnum = dp.value("Number").intValue();
            System.out.println( "Getting square of " + dpnum);
            TSSMeasure.think(2000);
            TSSMeasure.commandStart("square001", "getSquare",
                                    MST_WAITRESP);
            answer = square.getSquare(dpnum);
            TSSMeasure.commandEnd((short)TSS_CMD_STAT_PASS);
            System.out.println(answer);

            // Destroy EJB
            square.remove();
        }
        catch (RemoteException e) {
            System.err.println( "remoteException" + e.getMessage() );
            e.printStackTrace();
        }
        catch (NullPointerException e) {
            if (getHome() == null)
                System.err.println( "noHome" + e.getMessage() );
            else
                e.printStackTrace();
        }
        catch (Exception e) {
            System.err.println( "generalException" + e.getMessage() );
            e.printStackTrace();
        }
    }


    // Constructor

    public SquareClientTM_datapool() {
        super();
    }


    // Helper method to get the EJB's home.

    private static SquareHome getHome() {


       // Specify the name of the server so we can find the Square EJB.

        String homeName = "com/rational/square/SquareHome";


        // Specify the name of the host machine with the name server.
        // This example is intended to run locally.  Also, specify
        // the class name of the JNDI initial naming factory.

        Properties env = new Properties();
        env.put(Context.PROVIDER_URL, "iiop:///");
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejb.cb.runtime.CBCtxFactory");


        try {
            // The following is the simplest way to get the
             //InitialContext.

            InitialContext ctx = new InitialContext(env);
            java.lang.Object obj=ctx.lookup( homeName );

            if (obj == null) {
               System.out.println( "ctx.lookup returned null object" );
                return null;  // fail
            }
```

```
                return ((SquareHome)
                  javax.rmi.PortableRemoteObject.narrow(obj,
                  com.rational.square.SquareHome.class));

            } catch (javax.naming.NamingException e) {
                e.printStackTrace();
                return null;
            }
        }

        public static class DatapoolConfig extends DatapoolInfo {
            public DatapoolConfig() {
                setDatapoolName("squaredp");
                setDatapoolAccessFlags(TSS_DP_WRAP |
                                       TSS_DP_SEQUENTIAL |
                                       TSS_DP_SHARED);
            }
        }

        public static void main(String args[]) {
            SquareClientTM_datapool sctm = new SquareClientTM_datapool();
            sctm.testMain(args);
        }
    }
```

Synchronization Point Configuration

A test script that uses a synchronization point must include, somewhere in its body, a block of code such as the following:

```
public static class SyncPointConfig extends SyncPointInfo {
        public SyncPointConfig() {
            setSyncPointNames(java.lang.String[] points);
        }
    }
```

The *points* argument of setSyncPointNames() — a method of SyncPointInfo — is an array containing the names of one or more synchronization points. Add to this array the name of each synchronization point in the script that you specified with TSSSync.syncPoint().

The following is an example of a Java program that uses a synchronization point named square_syncpoint. Relevant lines apear in bold.

```
/**
 * SquareClientTM demonstrates an EJB client that can be executed
 * from Rational Suite TestStudio using TestManager.
 *
 */

// EJB itself
```

```
import com.rational.square.Square;
import com.rational.square.SquareHome;

// Misc
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.ListResourceBundle;
import java.rmi.RemoteException;

// JNDI-related
import javax.naming.Context;
import javax.naming.InitialContext;

// TestManager
import com.rational.test.tss.*;


// Java test scripts must extend the TestScript interface.

public class SquareClientTM_syncpoint
    extends com.rational.test.tss.TestScript {

    public void testMain(String[] args) {
        try {

            // Create EJB
            TSSMeasure.commandStart("home001", "getHome",
                                     MST_XCLNTCONN);
            Square square = getHome().create();
            TSSMeasure.commandEnd((short)TSS_CMD_STAT_PASS);

            // Call Square method
            System.out.println( "Getting square of 123" );
            long answer = 0;
            TSSSync.syncPoint("square_syncpoint");
            TSSMeasure.think(2000);
            TSSMeasure.commandStart("square001", "getSquare",
                                     MST_WAITRESP);
            answer = square.getSquare(123);
            TSSMeasure.commandEnd((short)TSS_CMD_STAT_PASS);
            System.out.println(answer);

            // Destroy EJB
            square.remove();
        }
        catch (RemoteException e) {
            System.err.println( "remoteException" + e.getMessage() );
            e.printStackTrace();
        }
        catch (NullPointerException e) {
            if (getHome() == null)
                System.err.println( "noHome" + e.getMessage() );
            else
                e.printStackTrace();
        }
        catch (Exception e) {
            System.err.println( "generalException" + e.getMessage() );
            e.printStackTrace();
        }
    }


    // Constructor

    public SquareClientTM_syncpoint() {
        super();
    }


    // Helper method to get the EJB's home.

    private static SquareHome getHome() {


       // Specify the name of the server so we can find the Square EJB.

        String homeName = "com/rational/square/SquareHome";


        // Specify the name of the host machine with the name server.
        // This example is intended to run locally.  Also, specify
        // the class name of the JNDI initial naming factory.

        Properties env = new Properties();
        env.put(Context.PROVIDER_URL, "iiop:///");
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejb.cb.runtime.CBCtxFactory");


        try {
            // The following is the simplest way to get the
             //InitialContext.

            InitialContext ctx = new InitialContext(env);
            java.lang.Object obj=ctx.lookup( homeName );

            if (obj == null) {
               System.out.println( "ctx.lookup returned null object" );
                return null;  // fail
            }

            return ((SquareHome)
             javax.rmi.PortableRemoteObject.narrow(obj,
                             com.rational.square.SquareHome.class));

        } catch (javax.naming.NamingException e) {
            e.printStackTrace();
            return null;
        }
    }

    public static class SyncPointConfig extends SyncPointInfo {
        public SyncPointConfig() {
            String points[] = {
                "square_syncpoint"};
            setSyncPointNames(points);
        }
    }

    public static void main(String args[]) {
       SquareClientTM_syncpoint sctm = new SquareClientTM_syncpoint();
        sctm.testMain(args);
    }
}
```

Shared Variable Configuration

A test script that uses a shared variable must include, somewhere in its body, a block of code such as the following:

```
public static class SharedVarConfig extends SharedVarInfo {
        public SharedVarConfig() {
        setSharedVarNames(java.lang.String[] sv);
        }
    }
```

The *sv* argument of setSharedVarNames() — a method of SharedVarInfo — is an array containing the names of one or more shared variables. Add to this array the name of each shared variable in the script that you specified with one of the shared variable methods.

The following is an example of a Java program that uses a shared variable named square_number. Relevant lines apear in bold.

```
/**
* SquareClientTM demonstrates an EJB client that can be executed
 * from Rational Suite TestStudio using TestManager.
 *
 */

// EJB itself
import com.rational.square.Square;
import com.rational.square.SquareHome;

// Misc
import java.util.Properties;
import java.util.ResourceBundle;
import java.util.ListResourceBundle;
import java.rmi.RemoteException;

// JNDI-related
import javax.naming.Context;
import javax.naming.InitialContext;

// TestManager
import com.rational.test.tss.*;


// Java test scripts must extend the TestScript interface.

public class SquareClientTM_sharedvar
    extends com.rational.test.tss.TestScript {

    public void testMain(String[] args) {
        try {

            // Create EJB
            TSSMeasure.commandStart("home001", "getHome",
                                    MST_XCLNTCONN);
            Square square = getHome().create();
            TSSMeasure.commandEnd((short)TSS_CMD_STAT_PASS);

            // Call Square method
            long answer = 0;
            int retval;
            TSSInteger shval = new TSSInteger(0);
            try {
                retval = TSSync.sharedVarWait("square_number",
                                              1,
                                              1000000,
                                              0,
                                              30000,
                                              shval);
            } catch(TSSException e) {
                System.err.print(e);
                throw e;
            }
            System.out.println( "Getting square of " +
                                shval.getValue());
            TSSMeasure.think(2000);
            TSSMeasure.think(2000);
            TSSMeasure.commandStart("square001", "getSquare",
                                    MST_WAITRESP);
            answer = square.getSquare(shval.getValue());
            TSSMeasure.commandEnd((short)TSS_CMD_STAT_PASS);
            System.out.println(answer);

            // Destroy EJB
            square.remove();
        }
        catch (RemoteException e) {
            System.err.println( "remoteException" + e.getMessage() );
            e.printStackTrace();
        }
        catch (NullPointerException e) {
            if (getHome() == null)
                System.err.println( "noHome" + e.getMessage() );
            else
                e.printStackTrace();
        }
        catch (Exception e) {
            System.err.println( "generalException" + e.getMessage() );
            e.printStackTrace();
        }
    }


    // Constructor

    public SquareClientTM_sharedvar() {
        super();
    }


    // Helper method to get the EJB's home.

    private static SquareHome getHome() {


       // Specify the name of the server so we can find the Square EJB.

        String homeName = "com/rational/square/SquareHome";


        // Specify the name of the host machine with the name server.
        // This example is intended to run locally.  Also, specify
        // the class name of the JNDI initial naming factory.

        Properties env = new Properties();
        env.put(Context.PROVIDER_URL, "iiop:///");
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejb.cb.runtime.CBCtxFactory");


        try {
            // The following is the simplest way to get the
            // InitialContext.

            InitialContext ctx = new InitialContext(env);
            java.lang.Object obj=ctx.lookup( homeName );

            if (obj == null) {
               System.out.println( "ctx.lookup returned null object" );
                return null;  // fail
            }
            return ((SquareHome)
                     javax.rmi.PortableRemoteObject.narrow(obj,
                        com.rational.square.SquareHome.class));

        } catch (javax.naming.NamingException e) {
            e.printStackTrace();
            return null;
        }
    }

    public static class SharedVarConfig extends SharedVarInfo {
        public SharedVarConfig() {
            String sv[] = {
                "square_number"};
            setSharedVarNames(sv);
        }
    }

    public static void main(String args[]) {
        SquareClientTM_datapool sctm = new SquareClientTM_datapool();
        sctm.testMain(args);
    }
}
```

Java Support Classes

B

This appendix shows the source code for some Java support classes. They are not likely to be used directly or independently in test scripts, but they are used by several methods documented in this manual.

TSSNamedValue

`TSSNamedValue` is defined as follows:

```
package com.rational.test.tss;
public class TSSNamedValue
{
   public String name;
   public String value;
}
```

DatapoolValue

Used by *TSSDatapool.value()* on page 30, the `DatapoolValue` support class converts the data stored in datapools to an appropriate Java type.

```
package com.rational.test.tss;

/**
 *
 * DatapoolValue is returned from the TSSDatapool.value method.
 * Methods were copied from Component Test implementation.
 *
 * @author Sonny Pak
 * @version 1.0, 29-Jun-2000
 *
 * Modified:
 *
 *           Copyright (C) Rational Software Corporation, 2000
 *                             ALL RIGHTS RESERVED
 *
 */

public class DatapoolValue {

    private String value;

    DatapoolValue(String v) {
        value = v;
    }
    /*
     *
     * getBigDecimal
     *
     * @return java.math.BigDecimal
     * @excpetion java.lang.Exception The specified exception occurred
during the conversion attempt.
     *
     */
    public java.math.BigDecimal getBigDecimal() throws
java.lang.Exception {
   java.math.BigDecimal bigDecimalVal = null;
   try {
            java.math.BigDecimal bigDecimal = new
java.math.BigDecimal(value);
            bigDecimalVal = bigDecimal;
   }
   catch (java.lang.NumberFormatException nfe) {
           StringBuffer buf = new StringBuffer("");
           buf.append("DatapoolValue.");
           buf.append("getBigDecimal()");
           buf.append(" failed to convert the following value: ");
```

```
          buf.append(value);
          java.lang.NumberFormatException newExc = new
java.lang.NumberFormatException(buf.toString());
          throw newExc;
   }
   catch (Exception e) {
           throw e;
   }
   return bigDecimalVal;
    }
    /*
     *
     * booleanValue
     *
     * @return boolean
     * @exception java.lang.Exception This method throws
java.lang.Exception if the conversion attempt fails.
     *
     */
   public boolean booleanValue() throws java.lang.Exception {
   boolean bVal = false;

   // No special error handling because Boolean(string) constructor
can't throw an exception.
   try {
           Boolean b = new Boolean(value);
           bVal = b.booleanValue();
        } catch (Exception e) {
           throw e;
   }
   return bVal;
    }
    /*
     *
     * floatValue
     *
     * @return float
     * @exception java.lang.Exception This method throws an exception
if the conversion fails.
    * @exception java.lang.NumberFormatException This method throws an
exception if conversion fails.
     *
     */
    public float floatValue() throws java.lang.Exception,
        java.lang.NumberFormatException {
   float fVal = 0;
   try {
           Float f = new Float(value);
           fVal = f.floatValue();
   }
   catch (java.lang.NumberFormatException nfe) {
          StringBuffer buf = new StringBuffer("");
          buf.append("DatapoolValue.");
          buf.append("floatValue()");
```

```
            buf.append(" failed to convert the following value: ");
            buf.append(value);
            java.lang.NumberFormatException newExc = new
java.lang.NumberFormatException(buf.toString());
            throw newExc;
   }
   catch (Exception e) {
             throw e;
   }
   return fVal;
     }
     /*
      *
      * intValue
      *
      * @return int
      * @exception java.lang.Exception This method throws an exception
if the conversion fails.
     * @exception java.lang.NumberFormatException This method throws an
exception if conversion fails.
      *
      */
     public int intValue() throws java.lang.Exception,
          java.lang.NumberFormatException {
   int iVal = 0;
   try {
             Integer i = new Integer(value);
             iVal = i.intValue();
   }
   catch (java.lang.NumberFormatException nfe) {
            StringBuffer buf = new StringBuffer("");
            buf.append("DatapoolValue.");
            buf.append("intValue()");
            buf.append(" failed to convert the following value: ");
            buf.append(value);
            java.lang.NumberFormatException newExc = new
java.lang.NumberFormatException(buf.toString());
            throw newExc;
   }
   catch (Exception e) {
             throw e;
   }
   return iVal;
     }
     /*
      *
      * longValue
      *
      * @return long
      * @exception java.lang.Exception This method throws an exception
if the conversion fails.
     * @exception java.lang.NumberFormatException This method throws an
exception if conversion fails.
      *
      */
     public long longValue() throws java.lang.Exception,
          java.lang.NumberFormatException {
    long lVal = 0;
    try {
             Long l = new Long(value);
             lVal = l.longValue();
    }
    catch (java.lang.NumberFormatException nfe) {
            StringBuffer buf = new StringBuffer("");
            buf.append("DatapoolValue.");
            buf.append("longValue()");
            buf.append(" failed to convert the following value: ");
            buf.append(value);
            java.lang.NumberFormatException newExc = new
java.lang.NumberFormatException(buf.toString());
            throw newExc;
    }
    catch (Exception e) {
             throw e;
    }
    return lVal;
     }
     /*
      *
      * byteValue
      *
      * @return byte
      * @exception java.lang.Exception This method throws an exception
if the conversion fails.
      *
      */
     public byte byteValue() throws java.lang.Exception {
    byte bt;
    try {
             Byte b = new Byte(value);
             bt = b.byteValue();
    }
    catch (java.lang.NumberFormatException nfe) {
            StringBuffer buf = new StringBuffer("");
            buf.append("DatapoolValue.");
            buf.append("byteValue()");
            buf.append(" failed to convert the following value: ");
            buf.append(value);
            java.lang.NumberFormatException newExc = new
java.lang.NumberFormatException(buf.toString());
            throw newExc;
    }
    catch (Exception e) {
             throw e;
    }
    return bt;
     }
     /*
      *
      * charValue
      *
      * @return char
      * @exception java.lang.Exception This method throws an exception
if the conversion fails.
      *
      */
     public char charValue() throws java.lang.Exception {
    char ch;
    try {
             ch = value.charAt(0);
    } catch (Exception e) {
        if ( value == null || value.length() == 0 )
        {
            StringBuffer buf = new StringBuffer("");
            buf.append("DatapoolValue.”");
            buf.append("charValue()");
            buf.append(" failed to convert empty string to char.");
            java.lang.Exception newExc = new
java.lang.Exception(buf.toString());
            throw newExc;
        }
        else
        {
            StringBuffer buf = new StringBuffer("");
            buf.append("DatapoolValue.");
            buf.append("charValue()");
            buf.append(" failed to convert.");
            java.lang.Exception newExc = new
java.lang.Exception(buf.toString());
            throw newExc;
        }
    }
    return ch;
     }
     /*
      *
      * shortValue
      *
      * @return short
      * @exception java.lang.Exception This method throws an exception
if the conversion fails.
     * @exception java.lang.NumberFormatException This method throws an
exception if conversion fails.
      *
      */
     public short shortValue() throws java.lang.Exception,
         java.lang.NumberFormatException {
    short sVal = 0;
    try {
             Short s = new Short(value);
             sVal = s.shortValue();
    }
```

```
   catch (java.lang.NumberFormatException nfe) {
           StringBuffer buf = new StringBuffer("");
           buf.append("DatapoolValue.");
           buf.append("shortValue()");
           buf.append(" failed to convert the following value: ");
           buf.append(value);
           java.lang.NumberFormatException newExc = new
java.lang.NumberFormatException(buf.toString());
           throw newExc;
   }
   catch (Exception e) {
            throw e;
   }
   return sVal;
    }
    /*
     *
     * toString
     *
     * @return String
     *
     */
    public String toString() {
        return value;
    }
}
```

TSSConstants

The constants used as arguments in a number of TSS methods are defined in class `TSSConstants`, which is shown below.

```
/*
 *
 * Java Test Script Services Constants class
 * Public constants for Test Script Services
 *
 * Interface is implemented in the TestScript class to inherit the
constants
 *
 * @author DuWayne Morris
 * @version 1.0, 20-June-2000
 *
 * Modified:
 *
 *            Copyright (C) Rational Software Corporation, 2000
 *                            ALL RIGHTS RESERVED
 *
 */

package com.rational.test.tss;


public interface TSSConstants {
/*
 * return codes for TSS functions
 */
   public static final int TSS_NOOP = 1;
   public static final int TSS_OK = 0;
   public static final int TSS_FAIL = -1;
   public static final int TSS_EOF = -2;
   public static final int TSS_NOSERVER = -3;
   public static final int TSS_INVALID= -4;
   public static final int TSS_SYSERROR = -5;
   public static final int TSS_NOTFOUND = -6;
   public static final int TSS_ABORT = -7;

/*
 * context keys
 */
   public static final int CTXT_workingDir = 0;
   public static final int CTXT_datapoolDir = 1;
    public static final int CTXT_timeZero = 2;
   public static final int CTXT_todZero = 3;
   public static final int CTXT_END = 4;

/*
 * datapool open flags
 */
   public static final int TSS_DP_RANDOM = 0x0001;

//#define TSS_DP_SEQUENTIAL(TSS_DP_RANDOM << 4)
   public static final int TSS_DP_SEQUENTIAL = 0x0010;

//#define TSS_DP_SHUFFLE(TSS_DP_RANDOM << 8)
   public static final int TSS_DP_SHUFFLE = 0x0100;

//#define TSS_DP_RANDOM_MASK(TSS_DP_RANDOM | TSS_DP_SEQUENTIAL |
TSS_DP_SHUFFLE)
   public static final int TSS_DP_RANDOM_MASK = 0x0111;

//#define TSS_DP_WRAP0x0002
   public static final int TSS_DP_WRAP = 0x0002;

///#define TSS_DP_NOWRAP(TSS_DP_WRAP << 4)
   public static final int TSS_DP_NOWRAP = 0x0020;

//#define TSS_DP_WRAP_MASK(TSS_DP_WRAP | TSS_DP_NOWRAP)
   public static final int TSS_DP_WRAP_MASK = 0x0022;

//#define TSS_DP_SHARED0x0004
   public static final int TSS_DP_SHARED = 0x0004;

//#define TSS_DP_PRIVATE(TSS_DP_SHARED << 4)
   public static final int TSS_DP_PRIVATE = 0x0040;

//#define TSS_DP_SHARED_MASK(TSS_DP_SHARED | TSS_DP_PRIVATE)
   public static final int TSS_DP_SHARED_MASK = 0x0044;

//#define TSS_DP_PERSIST0x0008
   public static final int TSS_DP_PERSIST = 0x0008;

//#define TSS_DP_NO_OPEN0x1000
   public static final int TSS_DP_NO_OPEN = 0x1000;

//#define TSS_DP_OPEN0x2000
   public static final int TSS_DP_OPEN = 0x2000;

/*
 * command_status flags
 */
   public static final short TSS_CMD_STAT_FAIL =0x00;
   public static final short TSS_CMD_STAT_PASS =0x01;
   public static final short TSS_CMD_STAT_WARN =0x02;
   public static final short TSS_CMD_STAT_INFO =0x04;


/*
 * log event result codes
 */
   public static final short TSS_LOG_RESULT_NONE=          0;
   public static final short TSS_LOG_RESULT_PASS=          1;
```

```
    public static final short TSS_LOG_RESULT_FAIL=          2;
    public static final short TSS_LOG_RESULT_WARN=   3;
    public static final short TSS_LOG_RESULT_STOPPED=   4;
    public static final short TSS_LOG_RESULT_INFO =   5;
    public static final short TSS_LOG_RESULT_COMPLETED =   6;
    public static final short TSS_LOG_UNEVALUATED =       7;
    public static final short TSS_LOG_RESULT_NOT_RUN =   8;



  /*
   * timer flags (boolean)
   */
    public static final int TSS_TIMER_KEEP = 0x00;
    public static final int TSS_TIMER_REMOVE =0x01;

  // EvarOp

    public static final int EVOP_eval = 0;
    public static final int EVOP_pop = 1;
     public static final int EVOP_push = 2;
    public static final int EVOP_reset = 3;
    public static final int EVOP_restore = 4;
    public static final int EVOP_save = 5;
    public static final int EVOP_set = 6;
    public static final int EVOP_END = 7;

  // EvarKey

    public static final int EVAR_Think_avg =0;
    public static final int EVAR_Think_sd=1;
    public static final int EVAR_Think_dist =2;
    public static final int EVAR_Think_def =3;
    public static final int EVAR_Typing_dly =4;
    public static final int EVAR_Line =5;
    public static final int EVAR_Parity =6;
    public static final int EVAR_Baud =7;
    public static final int EVAR_Charsize =8;
    public static final int EVAR_Stopbits =9;
    public static final int EVAR_Timeout_val =10;
    public static final int EVAR_Timeout_act =11;
    public static final int EVAR_Escape_seq =12;
    public static final int EVAR_Logout_seq =13;
    public static final int EVAR_Log_level =14;
    public static final int EVAR_Record_level =15;
    public static final int EVAR_Key_map =16;
    public static final int EVAR_Flow_control =17;
    public static final int EVAR_Mystack =18;
    public static final int EVAR_Modem_control =19;
    public static final int EVAR_Mysstack =20;
    public static final int EVAR_Mybstack =21;
    public static final int EVAR_Emulation =22;
    public static final int EVAR_Screen_mask =23;
    public static final int EVAR_Screen_match =24;
    public static final int EVAR_Request_match =25;
    public static final int EVAR_Think_max =26;
    public static final int EVAR_Image_info =27;
    public static final int EVAR_Image_path =28;
    public static final int EVAR_Check_unread =29;
    public static final int EVAR_Initial_dly_max =30;
    public static final int EVAR_Think_dly_scale =31;
    public static final int EVAR_Typing_dly_scale = 32;
    public static final int EVAR_Delay_dly_scale =33;
    public static final int EVAR_Timeout_scale =34;
    public static final int EVAR_Suspend_check =35;
    public static final int EVAR_Server_connection =36;
    public static final int EVAR_CS_blocksize =37;
    public static final int EVAR_Column_headers =38;
    public static final int EVAR_Table_boundaries =39;
    public static final int EVAR_Sqlexec_control =40;
    public static final int EVAR_Max_nrecv_saved =41;
    public static final int EVAR_Sqlexec_control_sybase = 42;
    public static final int EVAR_Sqlexec_control_oracle = 43;
    public static final int EVAR_Audit =44;
    public static final int EVAR_Geom_polyfill =45;
    public static final int EVAR_Think_cpu_threshold = 46;
    public static final int EVAR_Think_cpu_dly_scale = 47;
    public static final int EVAR_Sqlexec_control_sqlserver = 48;
    public static final int EVAR_Connect_retries =49;
     public static final int EVAR_Connect_retry_interval = 50;
    public static final int EVAR_Sqlnrecv_long =51;
    public static final int EVAR_Statement_id  =52;
    public static final int EVAR_Http_control =53;
    public static final int EVAR_Iiop_bind_modi =54;
    public static final int EVAR_Iiop_principal =55;
    public static final int EVAR_Line_speed =56;
    public static final int EVAR_Cursor_id =57;
    public static final int EVAR_Iiop_control =59;
    public static final int EVAR_END =60;

 // IVKey

    public static final int IV_fcs_ts =0;
    public static final int IV_lcs_ts =1;
    public static final int IV_fcr_ts =2;
    public static final int IV_lcr_ts =3;
    public static final int IV_lineno =4;
    public static final int IV_cmdcnt =5;
    public static final int IV_uid =6;
    public static final int IV_ncxmit =7;
    public static final int IV_ncrecv =8;
    public static final int IV_ncnull =9;
    public static final int IV_nusers =10;
    public static final int IV_nkxmit =11;
    public static final int IV_nrows =12;
    public static final int IV_ncols =13;
    public static final int IV_row =14;
    public static final int IV_col =15;
```

```
    public static final int IV_fs_ts =16;
    public static final int IV_ls_ts =17;
    public static final int IV_fr_ts =18;
    public static final int IV_lr_ts =19;
    public static final int IV_nxmit =20;
    public static final int IV_nrecv =21;
    public static final int IV_button_no =22;
    public static final int IV_fuxe_ts =23;
    public static final int IV_luxe_ts =24;
    public static final int IV_uxe_cnt =25;
    public static final int IV_ig_fs_ts =26;
    public static final int IV_ig_ls_ts =27;
    public static final int IV_ig_eot_ts =28;
    public static final int IV_prev_ig_fs_ts =29;
    public static final int IV_prev_ig_ls_ts =30;
    public static final int IV_npixels_act =31;
    public static final int IV_npixels_exp =32;
    public static final int IV_npixels_diff =33;
    public static final int IV_xwin_diff_level = 34;
    public static final int IV_screen =35;
    public static final int IV_error =36;
    public static final int IV_total_rows =37;
    public static final int IV_statement_id =38;
    public static final int IV_error_logs =39;
    public static final int IV_cursor_id =40;
    public static final int IV_fc_ts =41;
    public static final int IV_lc_ts =42;
    public static final int IV_total_nrecv =43;
    public static final int IV_error_type =44;
    public static final int IV_tux_tpurcode =45;
    public static final int IV_command =46;
    public static final int IV_response =47;
    public static final int IV_source_file =48;
    public static final int IV_task_file =49;
    public static final int IV_cmd_id =50;
    public static final int IV_mcommand =51;
    public static final int IV_alltext =52;
    public static final int IV_error_text =53;
    public static final int IV_column_headers =54;
    public static final int IV_total_response =55;
    public static final int IV_script =56;
    public static final int IV_version =57;
    public static final int IV_user_group =58;
    public static final int IV_host =59;
    public static final int IV_refURI =60;
    public static final int IV_END =61;


  /*
   * shvaradj
   */
    public static final int SHVADJ_none = 0;
    public static final int SHVADJ_pre_inc = ‘P’;
    public static final int SHVADJ_post_inc = ‘p’;
    public static final int SHVADJ_pre_dec = 'M';
    public static final int SHVADJ_post_dec = 'm';


  /*
   * shvarops
   */
    public static final int SHVOP_assign = '=';
    public static final int SHVOP_add = '+';
    public static final int SHVOP_subtract = '-';
    public static final int SHVOP_multiply = '*';
    public static final int SHVOP_divide = '/';
    public static final int SHVOP_modulo = '%';
    public static final int SHVOP_and = '&';
    public static final int SHVOP_or = '|';
    public static final int SHVOP_xor = '^';
    public static final int SHVOP_shiftleft = '<';
    public static final int SHVOP_shiftright = '>';


  // RunState declarations
  public static final int MST_UNDEF =0x0000;          /* user's
  micro_state is undefined */
  public static final int MST_INIT  =     0x0001;          /* ... doing
  start-up initialization */
  public static final int MST_GETTASK =   0x0002;          /* ... waiting
  for task assignment */
  public static final int MST_ITDLY   =   0x0003;          /* ...
  inter-task delay */
  public static final int MST_INITTASK =  0x0004;          /* ...
  initializing task */
  public static final int MST_USERCODE =  0x0005;          /* ... SQAVu
  user code */
  public static final int MST_THINK =     0x0006;          /* ... thinking
  */
  public static final int MST_TYPE  =     0x0007;          /* ... typing
  */
  public static final int MST_WAITRESP =  0x0008;          /* ... waiting
  for response */
  public static final int MST_DSPLYRESP = 0x0009;          /* ...
  displaying response */
  public static final int MST_PMATCH =    0x000A;          /* ... matching
  response (precv) */
  public static final int MST_DELAY  =    0x000B;          /* ... user
  requested delay() */
  public static final int MST_SHVBLCK =   0x000C;          /* ... blocked
  from shv access */
  public static final int MST_SHVWAIT =   0x000D;          /* ... user
  requested shv wait */
  public static final int MST_SUSPENDED = 0x000E;          /* ...
  suspended */
  public static final int MST_CLEANUP =   0x000F;          /* ... cleaning
  up */
  public static final int MST_EXITED  =   0x0010;          /* ... exited
```

```
*/
public static final int MST_XCLNTCONN = 0x0011;         /* ... waiting
on X client connection */
public static final int MST_WATCH =     0x0012;          /* ...
interactive -W watch/rerecord */
public static final int MST_SHVREAD =   0x0013;          /* ... V_VP:
reading shared variable */
public static final int MST_XWINDUMP =  0x0014;          /* ...
xwindow_diff dumping window */
public static final int MST_XWINCMP =   0x0015;          /* ...
xwindow_diff comparing windows */
public static final int MST_BUTTON =    0x0016;         /* ... X button
action */
public static final int MST_MOTION =    0x0017;         /* ... X motion
*/
public static final int MST_XQUERY =    0x0018;          /* ... X query
function */
public static final int MST_XSYNC  =    0x0019;          /* ... X sync
state during X query */
public static final int MST_XMOVEWIN =  0x001A;          /* ... X move
window */
public static final int MST_XCLNTDISC = 0x001B;         /* ... waiting
on X client disconnect */
public static final int MST_EXTERN_C =  0x001C;          /* ...
executing external C code */
public static final int MST_SQLEXEC  =  0x001D;          /* ...
executing SQL statements */
public static final int MST_SATEXEC  =  0x001E;          /* ...
executing satellite script */
public static final int MST_CPUDLY   =  0x001F;          /* ... cpu
delay */
public static final int MST_FIND     =  0x0020;          /* ...
find_text / find_point */
public static final int MST_TEST     =  0x0021;          /* ...
testcase, emulate */
public static final int MST_SEND     =  0x0022;          /* ...
http/socket send */
public static final int MST_TUXEDO   =  0x0023;          /* ... Tuxedo
execution */
public static final int MST_SQABASIC_CODE = 0x0024;      /* ... running
SQABasic code */
public static final int MST_EXITSQABASIC = 0x0025;       /* ... exited
SQABasic code */
public static final int MST_WAITOBJ =   0x0026;          /* ...
SQABasic: waiting for object  */
public static final int MST_STARTAPP =  0x0027;          /* ...
SQABasic: starting app */
public static final int MST_BIND   =    0x0028;          /* ...
iiop_bind in progress */
public static final int MST_IIOP_INVOKE = 0x0029;        /* ...
iiop_invoke in progress */
public static final int MST_SEND_DELAY =0x002A;          /* ...
line_speed delay in send */
public static final int MST_RECV_DELAY =0x002B;          /* ...
```

```
line_speed delay in recv */
public static final int MST_TRN_PACING =0x002C;          /* ...
transactor pacing delay */
public static final int MST_INCL  =     0x00FF;          /* mask
including above basic states */
public static final int N_MST_INCL =    0x2D;            /* number of
above basic states */

// same as MST_XCLNTCONN and MST_XCLNTDISC
public static final int MST_SQLCONN = 0x0011;           /* ...waiting
on SQL client connection*/
public static final int MST_SQLDISC = 0x001B;           /* ...waiting
on SQL client disconnect*/

// same as MST_XCLNTCONN and MST_XCLNTDISC
public static final int MST_HTTPCONN = 0x0011;          /* ...waiting
on http connection */
public static final int MST_HTTPDISC = 0x001B;          /* ...waiting
on http disconnect */

// same as MST_XCLNTCONN and
public static final int MST_SOCKCONN = 0x0011;          /* ...waiting
on socket connection */
public static final int MST_SOCKDISC = 0x001B;          /* ...waiting
on socket disconnect */


}
```

TSSInteger

The `TSSInteger` class defines the `Integer` argument type used by a number of methods.

```
/*
 *
 * Java Test Script Services Integer class
 * Public wrapper class for passing mutable integers in TSS calls
 *
 * @author DuWayne Morris
 * @version 1.0, 19-Oct-2000
 *
 * Modified:
 *
 *           Copyright (C) Rational Software Corporation, 2000
 *                               ALL RIGHTS RESERVED
 *
 */

package com.rational.test.tss;


public class TSSInteger
{
   private int internalValue;

   public TSSInteger(int iValue){
      internalValue = iValue;
   }
   public void setValue(int iValue)
   {
      internalValue = iValue;
   }

   public int getValue()
   {
      return internalValue;
   }
}
```

TSSException

On error, most methods throw TSSException. You call TSSException.getErrorCode() to get the error code. This class is shown below.

```
/*
 *
 * Java Test Script Services Exception class
 * Public methods correspond to published external TSS software C
interfaces.
 *
 * @author DuWayne Morris
 * @version 1.0, 29-June-2000
 *
 * Modified:
 *
 *           Copyright (C) Rational Software Corporation, 2000
 *                          ALL RIGHTS RESERVED
 *
 */
package com.rational.test.tss;

import com.rational.test.tss.*;

public class TSSException extends java.lang.Exception
{

   private int iErrRet;
   private int iErrorCode;

   private TSSException(int retCode, int errCode, String strError)
   {
      super(strError);
      iErrRet = retCode;
      iErrorCode = errCode;
   }
   // this is the “official” error
   // from calling errorDetail
   public int getErrorCode()
   {
      return iErrorCode;
   }

   // this is the return value from the original native
   // method call
   public int getReturnCode()
   {
      return iErrRet;
   }

   public static TSSException exception(int ret){
```

```
            StringBuffer strBuf = new StringBuffer("");


            int i = TSSUtility.errorDetail(strBuf);

            if (i != 0)
            {
               TSSException e = new TSSException(ret, i, strBuf.toString());

               return e;
            }
            return null;
         }
         public static TSSException exception(int ret, String str){


               TSSException e = new TSSException(ret, ret, str);

               return e;
         }
      }
```

CTutil Class Source Code C

The utility methods in the following sample code are called by several of the examples in *Implementing a New Verification Point* on page 161. The code for the `CTutil` class is included in this appendix.

The `CTutil` class is in `rational_ct.jar`.

```
public class CTutil
{
   public static boolean csvGetNextElement( StringBuffer bufCSV,
                  StringBuffer bufElement )
   {
      String sCSV = bufCSV.toString();
      int iCommaIndex = sCSV.indexOf(',');

      if (iCommaIndex == -1)
      {
         bufElement.insert(0, sCSV);
         bufElement.setLength(sCSV.length());
         bufCSV.setLength(0);
         return false;
      }
      else
      {
         bufElement.insert(0, sCSV.substring(0, iCommaIndex));
         bufElement.setLength(iCommaIndex);
         bufCSV.insert(0, sCSV.substring(iCommaIndex+1));
         bufCSV.setLength( sCSV.length() - iCommaIndex - 1 );
         return true;
      }
   }

   // This function reads an INI file and returns a hashtable.  The
   // hashtable maps strings (section names from the INI file) to
   // hashtables of those sections.  These section hastables map
   // strings (keys from the section) to strings (values from those
   // keys.) You can pass the hashtable constructed by this function
   // to readPrivateProfileString(), which returns values from
   // the .INI file.
public static Hashtable mapINIfile(InputStream in) throws IOException
   {
      BufferedReader brIn = new BufferedReader (
         new InputStreamReader ( in ));

      String sLine = "";
      String sKey = "";
      String sValue = "";
      String sSection = "";

      int iEquals = 0;

      Hashtable tblINI = new Hashtable();
      Hashtable tblSection = new Hashtable();

      // Read the file one line at a time.
      for ( sLine = brIn.readLine(); sLine != null &&
                      sLine.trim() != null;
           sLine = brIn.readLine() )
      {
         sLine = sLine.trim();
         if ( sLine.length() == 0 )
         {
            continue;
         }
         else if ( sLine.charAt(0) == '[' )
         {
            // This is a new category.  If this isn't the first one,
            // write the previous one into the hashtable.
            if ( !tblSection.isEmpty() )
            {
               tblINI.put(sSection, tblSection);
               tblSection = new Hashtable();
            }

            // Store the new Section name.
            sSection = sLine.substring(1, sLine.length()-1);
         }
         else
         {
            // Find the separator between the key and the value.
            iEquals = sLine.indexOf('=');

            if ( iEquals < 0 )
            {
               // The entry in the INI file doesn't match INI spec.
               // ignore it and continue reading the file.
               continue;
            }
            else if ( iEquals == 0 )
            {
               // There is no Key name.  Invalid INI format.
               // ignore and continue.
               continue;
            }
            else if ( iEquals == sLine.length()-1 )
            {
               // Key with no Value.  Set the Value to null.
               sKey = sLine.substring(0, iEquals);
               sValue = "";
```

```
            tblSection.put(sKey, sValue);
         }
         else
         {
            // Parse the line.
            sKey = sLine.substring(0, iEquals);
            sValue = sLine.substring(iEquals+1);

            // Add the entry to the table for this section.
            tblSection.put(sKey, sValue);
         }
      }
   }

   if ( !tblSection.isEmpty() )
   {
      tblINI.put(sSection, tblSection);
   }

   if ( !tblINI.isEmpty() )
      return tblINI;
   else
      return null;
}

public static String readPrivateProfileString( Hashtable tblMap,
            String sSection, String sKey )
{
   String sValue = "";
   Hashtable tblSection = (Hashtable) tblMap.get(sSection);

   if ( tblSection != null )
   {
      sValue = (String) tblSection.get(sKey);
      if ( sValue == null )
         sValue = "";
   }
   return sValue;
}
}
```

Index