Rational Test Script Services for Visual Basic

VERSION 2001A.04.00

PART NUMBER 800-024530-000

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 COM/DCOM 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 Visual Basic. Your Visual Basic test scripts can be used for both performance and functional testing.

Other Resources

- To access an HTML version of this manual, click **TSS for Visual Basic** 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 Visual Basic Test Script Services

Rational *Test Script Services* are testing services that you can add to your Visual Basic (VB) 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 Visual Basic COM/DCOM components.

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

- Rational TestManager
- Rational QualityArchitect

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 Visual Basic 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 Visual Basic scripts that contain measurement inputs, such as verification methods for validating the behavior of objects.
- TestManager can run your Visual Basic scripts with scripts of other types, such as Java, GUI, and VU scripts.

The Test Script Services used with TestManager are documented in *Test Script Services Reference* on page 15.

Using Test Script Services With Rational QualityArchitect

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

- *Verification Services* on page 113
- *Database Verification Point Reference* on page 125
- *Verification Point Framework Reference* on page 135
- *Implementing a New Verification Point* on page 171

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
Monitor	Monitor test script playback progress.	TestManager, QualityArchitect
Synchronization	Synchronize multiple virtual testers running on a single computer or across multiple computers.	TestManager

Category	Description	Commonly Used With
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 COM/DCOM 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 Visual Basic Test Scripts

A Visual Basic *test script* is a set of Visual Basic source files used for testing applications and components within the Rational test environment.

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

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

Creating Test Scripts

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

- Generate a script with Rational Robot.

  Rational Robot automatically generates Visual Basic test scripts from recorded COM/DCOM traffic.

- 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 Visual Basic script using a supported version of Visual Basic. See the Release Notes for supported versions.

Entry Points

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 10.

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 **VB 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 into the **Option name** and **Option value** columns:

Option name	Option value
Type	VB
Default Datastore	0

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

Test Script Components

A test script contains the following source file types:

Source file type	Description
ScriptName.vbp	Project file. You reference this file and its location from your Visual Basic IDE.
ScriptName.bas	Standard module. This file initiates execution of a .cls test script file by calling the .cls file.
ScriptName.cls	Class module (test script file).
ScriptName.res	Resource file. This is an optional file that contains information about datapool, synchronization point, and shared variable configuration. Resource files are compiled from .rc files.

When compiled through TestManager, Visual Basic test scripts are compiled as .dll files. When compiling a Rational QualityArchitect test script in Visual Basic, compile the test script as a .exe file.

Updating the Project File

The project file must contain specific references.

Robot automatically inserts the required references into the project file when it records a Visual Basic script.

If you are creating and coding a script manually, or if you are not using a QualityArchitect template that contains the required references, you must insert the references into the project file yourself.

To insert the references into the project file:

1. Click **Project > References**.
2. In the **Available References** list, select **TestServices 1.0**.
3. If you are creating a Rational QualityArchitect script, also select **Rational QualityArchitect Playback Type Library**.
4. Click **OK**.

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 one of the following:

- **VB - (Rational Test Datastore)**. Choose this type for scripts that were auto-generated by Robot.
- The name of the Visual Basic source folder that you created in section *Registering Test Script Source Folders* on page 6. Choose this type for test scripts that you manually created from your IDE.

Then select the script you want to open. TestManager checks the Windows Registry to find the IDE associated with the test script.

When you save a test script, the location where you store the script and any accompanying source files will vary, as follows:

- If you record a script with Rational Robot, Robot automatically stores the script in the TMS_Scripts folder of the current Rational project and datastore — for example:

  *ProjectName**DatastoreName*\DefaultTestScriptDatastore\TMS_Scripts

- 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 (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 and any related source 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.

Test Script Names

When you record a test script with Rational Robot, a .vbp, .cls, and .rc file is created for the script. Each file uses the name that you assign during the recording operation.

When the test script is compiled in TestManager, a prefix is affixed to the name. For information, see *Name of the Compiled File* on page 9.

The maximum name length of scripts stored inside the Rational datastore is 40 characters. Script names cannot contain spaces.

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 .dll against the .cls, .vbp, and .res files. If the .dll is out of date, TestManager compiles the script before running it.

To compile a script, TestManager locates the Visual Basic compiler 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.

Compiled test scripts are stored as follows:

- If you compile through TestManager or through the command-line Test Script Execution Engine (TSEE), the script is compiled to a .dll file. This file is stored in an exe subfolder below the location of the script file.
- If you compile through QualityArchitect, the script is compiled to an .exe file stored in the same folder as the script.

Name of the Compiled File

When you compile a QualityArchitect test script in Visual Basic, standard Visual Basic naming conventions apply to the resulting .exe file.

When your test script is compiled through TestManager, the .dll file is assigned a name according to the following format:

mmm_sss

In this format, *mmm* is the name of the computer where the script was recorded and generated, and *sss* is the user-assigned script name. For example, if you record a script on a computer named echo and you name the script Test1, the full name of the .dll file is as follows:

echo_Test1.dll

Name-Checking at Playback

Before TestManager plays back a compiled script with a name in the *mmm_sss* format, it checks that the name of the computer where the compiled script will run matches the computer name embedded in the .dll file name. If the names don't match, TestManager does not execute the script.

TestManager undertakes this name-checking to ensure that a script will be executed in the same COM environment in which it was recorded. For more information, see *Moving a Test Script to a Different Computer for Playback* on page 11.

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 Visual Basic.

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

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 Visual Basic scripts to a suite that also contains Java, 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 .cls test script that you want to run inside a suite must implement the `ITestInterface_TestMain` method. This method is the entry point for the class.

The following is an example of the method declaration:

```
Public Function ITestinterface_TestMain(
     Optional ByVal args As Variant) As Variant
```

Running Test Scripts Outside TestManager

How you run a test script from your IDE depends on whether you are using Rational QualityArchitect.

To run a QualityArchitect test script from Visual Basic, the module file (.bas file) must include a `Main` entry point, and `Main` must include a call to `ITestInterface_TestMain` in a .cls file.

To run a test script from your IDE when QualityArchitect is not involved, do the following:

1. With the test script open in Visual Basic, click **Run > Start**.

   The RTScript - Project Properties dialog box appears.

2. Click **Start Program**, and then click the browse button (containing three dots) to the right of **Start Program**.

3. Select Rational\Rational Test\rttsee.exe in the Rational installation directory — for example:

   C:\Program Files\Rational\Rational Test\rttsee.exe

4. After rttsee.exe, type a space and the following bold text:

   C:\Program Files\Rational\Rational Test\rttsee.exe **-e rttseavb ScriptName**

   In the example, *ScriptName* is the name of your Visual Basic test script.

5. Click **OK**.

Moving a Test Script to a Different Computer for Playback

When you record a test script with Rational Robot, the compiled .dll file includes the name of the computer from which the script was recorded. At playback, TestManager checks the name of the computer where playback is to occur against the name of the computer embedded in the .dll name. If the names do not match, TestManager does not compile and play back the .dll file.

TestManager undertakes this name-checking for the following reasons:

- Test scripts are machine-dependent.
- Test scripts must run within the same COM/DCOM environment in which they were recorded with Rational Robot.

If you want to run a test script on a different computer than the one where it was recorded, do the following:

- Copy the .wch file associated with the test script to the new computer, and then generate a script from the .wch file. Unlike script files, .wch files are not machine-dependent.

  For information about watch (.wch) files and generating scripts from .wch files, see the Rational Robot online Help.

- On the computer where the test script is to run, duplicate the COM/DCOM environment just as it existed on the computer where the test script was recorded.

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 30
- *TSSLog.Message* on page 32
- *TSSLog.TestCaseResult* on page 34
- *TSSMeasure.CommandEnd* on page 36
- *TSSMeasure.CommandStart* on page 38
- *TSSAdvanced.LogCommand* on page 96
- *TestLog.Message* on page 109
- *TestLog.WriteError* on page 110
- *TestLog.WriteStubError* on page 110
- *TestLog.WriteStubMessage* on page 111

For additional information about logging errors, see *Trapping Errors* on page 14.

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 Visual Basic, 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 69. Use the `TSSMonitor` methods to provide data that is used during TestManager's monitoring operations.
- *Synchronization Class* on page 78. Use the `TSSSync` methods to allow concurrently running scripts to share data.

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

Trapping Errors

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

If an error occurs and the script does not contain error 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 error.

If you want to trap certain errors, 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.

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 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–

```
Close() As Long
```

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 datapool identifier is invalid.

Comments

Only one open datapool at a time is supported. A `Close` is thus required between intervening `Open` 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.

```
Dim retVal As Long
Dim dp As New TSSDatapool
dp.Open "custdata"
retVal = dp.Close
```

See Also

`Open`

TSSDatapool.ColumnCount

Returns the number of columns in a datapool.

Syntax

```
ColumnCount () As Long
```

Return Value

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

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim columns as Long
Dim dp As New TSSDatapool
dp.Open "custdata"
columns = dp.ColumnCount
```

TSSDatapool.ColumnName

Gets the name of the specified datapool column.

Syntax

```
ColumnName (columnNumber As Long) As String
```

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.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim colName as String
Dim dp as New TSSDatapool
if (dp.Fetch = True) Then
   colName = dp.ColumnName 3
EndIf
```

TSSDatapool.Fetch

Moves the datapool cursor to the next row.

Syntax

```
Fetch () As Boolean
```

Return Value

This method returns `True` (success) or `False` (end-of-file).

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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` results in a runtime error.

Example

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

```
Dim retVal As Boolean
Dim dp As New TSSDatapool
dp.Open "custdata"
retVal = dp.Fetch
```

See Also

`Open`, `Seek`, `Value`

TSSDatapool.Open

Opens the named datapool and sets the row access order.

Syntax

```
Open (name As String, [accessFlags As Long], [overrides[] As
   NamedValue])
```

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.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.
- `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 errors and do not log it, TestManager will not be aware of the error 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.

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.

NamedValue is a dimensioned array of name/value pairs. For example, an array of 10 name/value pairs could be implemented as follows:

```
Dim NV(9,1) As String
NV(0,0)= "name1"
NV(0,1)= "value1"
NV(1,0)= "name2"
NV(1,1)= "value2"
...
```

Example

This example opens the datapool named custdata, with a modified row access.

```
Dim dp As New TSSDatapool
dp.Open "custdata",TSS_DP_SHUFFLE + TSS_DP_PERSIST
```

See Also

Close

TSSDatapool.Rewind

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

Syntax

Rewind ()

Error Codes

This method may generate one of the following error codes:

- 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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim dp As New TSSDatapool
dp.Open "custdata"
dp.Seek (2)
dp.Rewind
```

TSSDatapool.RowCount

Returns the number of rows in a datapool.

Syntax

```
RowCount() As Long
```

Return Value

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

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim rows as Long
Dim dp As New TSSDatapool
dp.Open "custdata"
rows = dp.RowCount
```

TSSDatapool.Search

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

Syntax

`Search` (*`keys`*`[] As NamedValue)`

Element	Description
`keys`	An array containing values to be searched for.

Error Codes

This method may generate one of the following error codes:

- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.
- `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 errors and do not log it, TestManager will not be aware of the error 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.

`NamedValue` is a dimensioned array of name/value pairs. For example, an array of 10 name/value pairs could be implemented as follows:

```
Dim NV(9,1) As String
NV(0,0)= "name1"
NV(0,1)= "value1"
NV(1,0)= "name2"
NV(1,1)= "value2"
...
```

Example

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

```
Dim toFind(0,1) As String
toFind(0,0)= "Last"
toFind(0,1)= "Doe"
Dim dp As New TSSDatapool
dp.Open "custdata"
if (dp.Fetch = True) Then
   dp.Search toFind
EndIf
```

TSSDatapool.Seek

Moves the datapool cursor forward.

Syntax

```
Seek(count As Long)
```

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

Return Value

Error Codes

This method may generate one of the following error codes:

- 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 errors and do not log it, TestManager will not be aware of the error 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.

An end-of-file (TSS_EOF) error results if cursor wrapping is disabled (by access flag TSS_DP_NOWRAP) and *count* moves the access row beyond the last row. If Value is then called, a runtime error occurs.

Example

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

```
Dim dp As 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

`Value` (*`columnName`* As String) As Variant

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.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.
- `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 errors and do not log it, TestManager will not be aware of the error 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.

Example

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

```
Dim colVal as Variant
Dim dp As New TSSDatapool
dp.Open "custdata"
if (dp.Fetch = True) Then
   colVal = dp.Value "Middle"
EndIf
```

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 42) 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 41) 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 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

```
Event (eventType As String, [result As Integer], [description
   As String], [property[] As NamedValue])
```

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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 41) or `EVAR_LogEvent_control` (page 42) environment variables. Alternatively, the logging preference can be set with the `EVAR_Log_level` (page 42) and `EVAR_Record_level` (page 43) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

`NamedValue` is a dimensioned array of name/value pairs. For example, an array of 10 name/value pairs could be implemented as follows:

```
Dim NV(9,1) As String
NV(0,0)= "name1"
NV(0,1)= "value1"
NV(1,0)= "name2"
NV(1,1)= "value2"
...
```

Example

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

```
Dim scriptProp (1,1) As String
scriptProp(0,0)= "ScriptName"
scriptProp(0,1)= "Login"
scriptProp(1,0)= "LineNumber"
scriptProp(1,1)= "1"
Dim log As New TSSLog
log.Event "Login Dialog",0,"Login script failed",scriptProp
```

TSSLog.Message

Logs a message.

Syntax

`Message` (*`message`* `As String, [`*`result`* `As Integer], [`*`description`* `As String])`

Element	Description
`message`	Specifies the string to log.

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	Specifies the string to be put in the entry's failure description field.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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 41) or EVAR_LogEvent_control (page 42) environment variables. Alternatively, the logging preference can be set with the EVAR_Log_level (page 42) and EVAR_Record_level (page 43) 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--`.

```
Dim log As New TSSLog
log.Message "--Beginning of timed block T1--"
```

TSSLog.TestCaseResult

Logs a test case result.

Syntax

```
TestCaseResult (testcase As String, [result As Integer],
   [description As String], [property[] As NamedValue])
```

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.

- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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 41) or EVAR_LogEvent_control (page 42) environment variables. Alternatively, the logging preference may be set by the EVAR_Log_level (page 42) and EVAR_Record_level (page 43) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

NamedValue is a dimensioned array of name/value pairs. For example, an array of 10 name/value pairs could be implemented as follows:

```
Dim NV(9,1) As String
NV(0,0)= "name1"
NV(0,1)= "value1"
NV(1,0)= "name2"
NV(1,1)= "value2"
...
```

Example

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

```
Dim loginResult(0,1) As String
loginResult(0,0)= "Result"
loginResult(0,1)= "OK"
Dim log As New TSSLog
log.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 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

```
CommandEnd ([result As Integer], [description As String],
   [starttime As Long], [endtime As Long], [logdata As String],
   [property[] As NamedValue])
```

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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 41) or `EVAR_LogEvent_control` (page 42) environment variables. Alternatively, the logging preference can be set with the `EVAR_Log_level` (page 42) and `EVAR_Record_level` (page 43) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

`NamedValue` is a dimensioned array of name/value pairs. For example, an array of 10 name/value pairs could be implemented as follows:

```
Dim NV(9,1) As String
NV(0,0)= "name1"
NV(0,1)= "value1"
NV(1,0)= "name2"
NV(1,1)= "value2"
...
```

Example

This example marks the end of the timed activity specified by the previous `CommandStart` call.

```
Dim measure As TSSMeasure
measure.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

`CommandStart``(`*`label`* `As String, `*`name`* `As String, `*`state`* `As Long)`

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.
state	The run state to log with the timed command. See the run state table starting on page 75.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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 75.

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.

```
Dim measure As TSSMeasure
measure.CommandStart "initTimer", "Login", MST_WAITRESP
```

See Also

CommandEnd, TSSAdvanced.LogCommand

TSSMeasure.EnvironmentOp

Sets a virtual tester environment variable.

Syntax

EnvironmemtOp (*envVar* As EvarKey, *envOp* As EvarOp, *envVal* As Variant)

Element	Description
envVar	The environment variable to operate on. Valid values are described in the environment variable table starting on page 41.
envOP	The operation to perform. Valid values are described in the environment operations table starting on page 47.
envVal	The value operated on as specified by *envOP* to produce the new value for *envVar*.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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

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.

`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.

Name	Type/Values/(default)	Contains
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.
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.

```
Dim measure As New TESTSERVICESLib.TSSMeasure
measure.EnvironmentOP EVAR_Suspend_check, EVAR_pop, "OFF"
'input emulation code
measure.EnvironmentOP EVAR_Suspend_check, EVAR_pop, "ON"
```

TSSMeasure.GetTime

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

Syntax

```
GetTime() As Long
```

Return Value

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

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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`*.

```
Dim etime As Long
Dim measure As New TSSMeasure
etime = measure.GetTime
```

TSSMeasure.InternalVarGet

Gets the value of an internal variable.

Syntax

```
InternalVarGet (internVar As IVKey, ivVal As Variant)
```

Element	Description
internVar	The internal variable to operate on. Valid values are described in the internal variables table on page 49.
ivVal	OUTPUT. The returned value of the specified *internVar*.

Error Codes

This method may generate one of the following error codes:

- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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 internal variables that can be entered with the *internVar* argument.

Variable	Contains
alltext	Response text up to the value of Max_nrecv_saved. The same as response.
cmd_id	The ID of the most recent emulation command.

Variable	Contains
cmdcnt	A running count of the number of emulation commands the script has executed.
col	The current column position (1-based) of the cursor (ASCII screen emulation variable).
column_headers	The two-line column header if Column_headers is ON; otherwise.
command	The text of the most recent emulation command.
cursor_id	The last cursor declared by sqldeclare_cursor or opened by sqlopen_cursor.
error	The status of the last emulation command. Most values for error are supplied by the server.
error_text	The full text of the error from the last emulation command. If error is 0,error_text returns . For an SQL database or TUXEDO error, the text is provided by the server.
error_type	If you are emulating a TUXEDO session and error is nonzero, 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 under test Error 6 Internal Error If you are emulating an IIOP session and error is nonzero, error_type contains one of the following values: 0 (no error) 1 IIOP_EXCEPTION_SYSTEM 2 IIOP_EXCEPTION_USER 3 IIOP_ERROR
fc_ts	The "first connect" timestamp for http_request and sock_connect.
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, fr_ts is set to the time the SQL database server responded to the SQL statement.

Variable	Contains
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.
host	The host name of the computer on which the script is running.
lc_ts	The "last connect" timestamp for http_request and sock_connect.
lineno	The line number in source_file of the previously executed emulation command.
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, lr_ts is set to the time the SQL database server responded to the SQL statement.
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.
mcommand	The actual (mapped) sequence of characters submitted to the application under test by the most recent send or msend command. For send commands, mcommand is always equivalent to command.
ncnull	The number of null characters in an application response examined by the previous receive command in attempting to match this response.
ncols	The number of columns in the current screen (ASCII screen emulation variable).
ncrecv	The total number of non-null characters from an application response examined by the previous receive command in attempting to match this response.
ncxmit	The total number of characters transmitted to the application by the previous send or msend command.
nkxmit	The total number of "keystrokes" transmitted to the application by the previous send or msend command. For send commands, nkxmit is always equivalent toncxmit.
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.
nrows	The number of rows in the current screen (ASCII screen emulation variable).

Variable	Contains
nusers	The number of total virtual testers in the current TestManager session.
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.
response	Same asrow.
row	The current row position (1-based) of the cursor (ASCII screen emulation variable).
script	The name of the script currently being executed.
source_file	The name of the file that was the source for the portion of the script being executed.
statement_id	The value assigned as the prepared statement ID, which is returned by sqlprepare and sqlalloc_statement.
total_nrecv	The total number of bytes received for all HTTP and socket receive emulation commands issued on a particular connection.
total_rows	Set to the number of rows processed by the SQL statements. If the SQL statements do not affect any rows, total_rows is set to 0. If the SQL statements return row results, total_rows is set to 0 by sqlexec, then incremented by sqlnrecv as the row results are retrieved.
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.
uid	The numeric ID of the current virtual tester.
user_group	The name of the user group (from the suite) of the virtual tester running the script.
version	The full version string of TestManager (for example, 7.5.0.1045).

Example

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

```
Dim measure As New TSSMeasure
measure.InternalVarGet IV_error,IVVal
```

TSSMeasure.Think

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

Syntax

Think ([*thinkAverage* As Long])

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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 40.

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.

```
Dim measure As New TSSMeasure
measure.Think
```

See Also

TSSAdvanced.ThinkTime

TSSMeasure.TimerStart

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

Syntax

```
TimerStart ([label As String], [timeStamp As Long])
```

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim measure As New TSSMeasure
measure.TimerStart "event1"
'actions to be timed
measure.TimerStop "event1"
```

See Also

`TimerStop`

TSSMeasure.TimerStop

Marks the end of a block of timed actions.

Syntax

`TimerStop` (*`label`* `As String, [`*`timeStamp`* `As Long], [`*`rmFlag`* `As Long])`

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 0(default) to stop the timer without removing it; otherwise, specify as non-zero. A timer that is not removed can be stopped multiple times in order to measure intervals comprising this timed event.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim measure As New TSSMeasure
measure.TimerStart()
'actions to be timed
measure.TimerStop "event1"
'other actions to be timed
measure.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 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

```
Delay (msecs As Long)
```

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

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim util As New TSSUtility
util.Delay(10)
```

TSSUtility.ErrorDetail

Retrieves error information about a failure.

Syntax

```
ErrorDetail (errorText As String) As Long
```

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.

Error Codes

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*.

- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `ERROR_NO_ERROR_MESSAGE`. An attempt was made to fetch a non-existent message.
- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.
- `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 errors and do not log it, TestManager will not be aware of the error 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 datapool and, if there is an error, displays the associated error message text.

```
Dim fetchRet As Boolean
Dim errorText As String
Dim dp As New TSSDatapool
Dim utility As New TSSUtility
dp.Open "custdata"
fetchRet = dp.Fetch
if (fetchRet = False) Then
   utility.ErrorDetail(errorText)
   MsgBox "Datapool fetch failed:", &errorText
EndIf
```

TSSUtility.GetScriptOption

Gets the value of a script playback option.

Syntax

GetScriptOption(*optionName* As String) As String

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim optVal As Variant
Dim util As New TSSUtility
optVal = util.GetScriptOption "repeat_count"
```

TSSUtility.GetTestCaseConfigurationName

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

Syntax

```
GetTestCaseConfigurationName() As String
```

Error Codes

This method may generate one of the following error codes:

- ERROR_CREATE_SAVE_ARRAY. An attempt to create or destroy a SAFEARRAY failed (which is likely a system rather than a script error).
- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.
- 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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim tcConfig As String
Dim util As New TSSUtility
tcConfig = util.GetTestCaseConfigurationName
```

TSSUtility.GetTestCaseName

Gets the name of the test case in use.

Syntax

```
GetTestCaseName() As String
```

Return Value

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

Error Codes

This method may generate one of the following error codes:

- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.
- `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 errors and do not log it, TestManager will not be aware of the error 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`.

```
Dim tcName As String
Dim util As New TSSUtility
tcName = util.GetTestCaseName
```

TSSUtility.NegExp

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

Syntax

`NegExp` (*`mean`* `As Long`) `As Long`

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.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim next As Long
Dim util As New TSSUtility
util.SeedRand 10
next = util.NegExp(10)
```

See Also

`Rand, SeedRand, Uniform`

TSSUtility.Rand

Gets the next random number.

Syntax

```
Rand() As Long
```

Return Value

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

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim next as Long
Dim util As New TSSUtility
next = util.Rand()
```

See Also

`SeedRand, NegExp, Uniform`

TSSUtility.SeedRand

Seeds the random number generator.

Syntax

`SeedRand` (*`seed`* `As Long`)

Element	Description
`seed`	The base integer.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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:

```
Dim util As New TSSUtility
util.SeedRand(10)
```

See Also

`Rand, NegExp, Uniform`

TSSUtility.StdErrPrint

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

Syntax

`StdErrPrint` (*`message`* `As String`)

Element	Description
message	The string to print.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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`.

```
Dim util As TSSUtility
util.StdErrPrint "Login failed"
```

See Also

`TSSUtility.StdErrPrint`

TSSUtility.StdOutPrint

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

Syntax

`StdOutPrint` (*`message`* `As String`)

Element	Description
message	The string to print.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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`.

```
Dim util As TSSUtility
util.StdOutPrint "Login successful"
```

See Also

`TSSUtility.StdErrPrint`

TSSUtility.Uniform

Gets the next uniformly distributed random number.

Syntax

`Uniform` (*`low`* `As Long,` *`high`* `As Long) As Long`

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.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim next As Long
Dim util As New TSSUtility
util.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 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

```
Display (message As String)
```

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

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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`.

```
Dim monitor As Net TSSMonitor
monitor.Display "Beginning transaction"
```

TSSMonitor.PositionGet

Gets the test script file name or line number position.

Syntax

`PositionGet`(*`srcFile`* As String , *`lineNumber`* As Long)

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`*.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim scriptFile As String
Dim lineNumber As Long
Dim monitor as New TSSMonitor
monitor.PositionGet scriptFile, lineNumber
```

See Also

`PositionSet`

TSSMonitor.PositionSet

Sets the test script file name or line number position.

Syntax

`PositionSet` (`[`*`srcFile`* `As String], [`*`lineNumber`* `As Long]`)

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.

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim monitor As New TSSMonitor
monitor.PositionSet "checkLogin",0
```

See Also

PositionSet

TSSMonitor.ReportCommandStatus

Reports the runtime status of a command.

Syntax

ReportCommandStatus (*status* As Long)

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.

Error Codes

This method may generate one of the following error codes:

- 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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim monitor as New TSSMonitor
monitor.ReportCommandStatus TSS_CMD_STAT_FAIL
```

TSSMonitor.RunStateGet

Gets the run state.

Syntax

```
RunStateGet() As Long
```

Return Value

On success, this method returns one of the run state values listed in the run state table starting on page 75.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim orig As Long
Dim monitor As New TSSMonitor
orig = monitor.RunStateGet
```

See Also

`RunStateSet`

TSSMonitor.RunStateSet

Sets the run state.

Syntax

```
RunStateSet (state As Long)
```

Element	Description
state	The run state to set. Enter one of the run state values listed in the run state table starting on page 75.

Error Codes

This method may generate one of the following error codes:

- `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 errors and do not log it, TestManager will not be aware of the error 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_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.

```
Dim monitor As New TSSMonitor
monitor.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 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

```
SharedVarAssign (name As String, value As Long, [op As Long]) As
   Long
```

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 (default) ▪ SHVOP_add ▪ SHVOP_subtract ▪ SHVOP_multliply ▪ SHVOP_divide ▪ SHVOP_modulo ▪ SHVOP_and ▪ SHVOP_or ▪ SHVOP_xor ▪ SHVOP_shiftleft ▪ SHVOP_shiftright

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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 adds 5 to the value of the shared variable `lineCounter` and puts the new value of `lineCounter` in `returnval`.

```
Dim returnVal as Long
Dim sync As New TSSSync
returnVal = sync.SharedVarAssign "lineCounter", 5, SHVOP_add
```

See Also

`SharedVarEval, SharedVarWait`

TSSSync.SharedVarEval

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

Syntax

`SharedVarEval` (*`name`* `As String, ` *`value`* `As Long, [`*`op`* `As Long]) As Long`

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.
`op`	Increment/decrement operator for the returned value: Can be one of the following: ▪ `SHVADJ_none` (default) ▪ `SHVADJ_pre_inc` ▪ `SHVADJ_post_inc` ▪ `SHVADJ_pre_dec` ▪ `SHVADJ_post_dec`

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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`.

```
Dim val, retVal as Long
Dim sync As New TSSSync
retVal = sync.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

```
SharedVarWait (name As String, min As Long, [max As Long],
   [adjust As Long], [timeout As Long], [returnVal As Long]) As
   Long
```

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*.

Return Value

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

Error Codes

This method may generate one of the following error codes:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- 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 errors and do not log it, TestManager will not be aware of the error 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.)

```
Dim retVal, svVal As Long
svVal = 0
Dim sync As New TSSSync
retVal = sync.SharedVarWait "inProgress",10,20,0,60000,svVal
```

See Also

`SharedVarAssign, SharedVarEval`

TSSSync.SyncPoint

Puts a synchronization point in a script.

Syntax

`SyncPoint` (*`label`* `As String`)

Element	Description
`label`	The name of the synchronization point.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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 Appendix A.

Example

This example creates a sync point named `BlockUntilSaveComplete`.

```
Dim sync As New TSSSync
sync.SyncPoint "BlockUntilSaveComplete"
```

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 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

```
Connect (host As String, port As Integer, id As Long) As Long
```

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

This method exits with one of the following results:

- ERROR_CONVERT_BSTR. An encountered string cannot be converted.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- TSS_OK. Success.
- TSS_NOOP. A connection and ID had already been established for this execution thread.
- TSS_NOSERVER. No TSS server was listening on *port*.
- TSS_SYSERROR. A system error occurred. Call ErrorDetail for information.
- 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.

```
Dim session as New TSSSession
session.Connect "192.36.25.107",port ,0
```

See Also

ServerStart

TSSSession.Context

Passes context information to a TSS server.

Syntax

Context (*ctx* As ContextKey, *value* As String) As Long

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

This method exits with one of the following results:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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.

```
Dim session As New TSSSession
session.Context CTXT_workingDir,"C:\temp"
```

TSSSession.Disconnect

Disconnects from a TSS proxy server.

Syntax

```
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.

```
Dim session as New TSSSession
session.Disconnect
```

TSSSession.ServerStart

Starts a TSS proxy server.

Syntax

```
ServerStart (port As Integer) As Long
```

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*.

Return Value

This method exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOOP`. A TSS server was already listening on *port*.
- `TSS_NOSERVER`. Start failure. Call `ErrorDetail` for information.
- `TSS_SYSERROR`. A system error occurred. Call `ErrorDetail` for information.
- `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*.

```
Dim port As Long
port = 0
Dim session as New TSSSession
session.ServerStart port
```

See Also

`ServerStop`

TSSSession.ServerStop

Stops a TSS proxy server.

Syntax

```
ServerStop (port As Integer) As Long
```

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

Return Value

This method exits with one of the following results:

- `TSS_OK`. Success.
- `TSS_NOOP`. No TSS server was listening on *port*.
- `TSS_INVALID`. No proxy TSS server was found or stopped.
- `TSS_SYSERROR`. A system error occurred. Call `ErrorDetail` for information.
- `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`.

```
session.ServerStop port
```

See Also

`ServerStart`

TSSSession.Shutdown

Stops logging and initializes TSS.

Syntax

```
Shutdown()
```

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.

```
Dim session As New TSSSession
...
session.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 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

`InternalVarSet` (*`internVar`* `As IVKey,` *`ivVal`* `As Variant`)

Element	Description
`internVar`	The internal variable to operate on. Internal variables and their values are listed in the table starting on page 49.
`ivVal`	The new value for *`internVar`*.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `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 errors and do not log it, TestManager will not be aware of the error 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.

```
Dim advanced TSSAdvanced
advanced.InternalVarSet IV_cmdcnt,0
```

See Also

`TSSMeasure.InternalVarGet`

TSSAdvanced.LogCommand

Logs a command event.

Syntax

```
LogCommand (name As String, label As String, [result As
   Integer], [description As String], [starttime As Long],
   [endtime As Long], [logdata As String], [property[] As
   NamedValue])
```

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.

Error Codes

This method may generate one of the following error codes:

- `ERROR_CONVERT_BSTR`. An encountered string cannot be converted.
- `ERROR_INVALID_PARM`. A required argument is missing or invalid.
- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.
- `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 errors and do not log it, TestManager will not be aware of the error 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 41) or `EVAR_LogEvent_control` (page 42) environment variables. Alternatively, the logging preference may be set with the `EVAR_Log_level` (page 42) and `EVAR_Record_level` (page 43) environment variables. The TSS_LOG_RESULT_STOPPED, TSS_LOG_RESULT_COMPLETED, and TSS_LOG_RESULT_UNEVALUATED preferences are intended for internal use.

NamedValue is a dimensioned array of name/value pairs. For example, an array of 10 name/value pairs could be implemented as follows:

```
Dim NV(9,1) As String
NV(0,0)= "name1"
NV(0,1)= "value1"
NV(1,0)= "name2"
NV(1,1)= "value2"
...
```

Example

This example logs a message for a login script.

```
Dim advanced As TSSAdvanced
advanced.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

`ThinkTime` `([`*`thinkAverage`* `As Long]) As Long`

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, this method returns a calculated think-time average. A negative exit value indicates an error. Callfor more information

Error Codes

This method may generate the following error code.

- `ERROR_INVALID_PARM`. A required argument is missing or invalid.

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‘
Dim pause, iv As Long
Dim advanced As New TSSAdvanced
Dim measure As New TSSMeasure
iv = measure.GetTime
advanced.InternalVarSet IV_fcs_ts, iv
advanced.InternalVarSet IV_lcs_ts, iv
advanced.InternalVarSet IV_fcr_ts, iv
advanced.InternalVarSet IV_lcr_ts, iv
pause = advanced.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 102

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

- *TestLog Class* on page 106

  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 QualityArchitect.

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 `expectedError` are the corresponding output values:

amount	interest	months	expectedReturn	expectedError
100000	0.0700	240	775.30	
125000	00725	300		-1
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

The items in this class are members of `RTCOMVPLib.LookupTable`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this class.

LookUpTable Example

The following example opens the lookup table `_Account_Info_L` and searches for the lookup table values *`ParamValues`* within the column names *`ParamNames`*. This example also uses the `TestLog` methods `WriteStubMessage` and `WriteStubError` to log status and error information about the lookup table operations.

```
Private Function Info_LookUp(ParamNames As Variant, AccountInfo()
        As Variant) As String
    On Error GoTo ErrorHandler_Info_LookUp

    Dim luTable As LookupTable
    Dim tLog As TestLog
    Dim ParamValues As Variant
    Set luTable = New LookupTable
    Set tLog = New TestLog

    Err.Clear

    ' For complex data types, code below will not be sufficient.
    ' You will need to add code to generate a meaningful lookup call.
    ParamValues = Array(CStr(AccountInfo))

    luTable.Open "_Account_Info_L"

    ' For complex data types, code below will need to be modified.
    tLog.WriteStubMessage "Account stub, Info method. ",
           "Entered with following values: " + CStr(AccountInfo)

    If luTable.Search(ParamNames, ParamValues) Then
        Dim lErr As Long

        lErr = luTable.ExpectedException
        If lErr Then
            tLog.WriteStubMessage "Account stub, Info method. ",
                   "Raising error: " + CStr(lErr)
            On Error GoTo RaiseError_Info_LookUp
            Err.Raise lErr, "RQA",
                   "Error raised from stub for Account.Info"
            On Error GoTo ErrorHandler_Info_LookUp
```

```
            Else
                  'For complex return types, code below will not be
                  ' sufficient. You will need to add code to generate
                  ' a meaningful return value.
                  Info_LookUp = luTable.ReturnValue
            End If
        Else
            'For complex data types, code below will need to be Modified.
            Err.Raise 23, "RQA", "Unique entry for input values, "
                  + "CStr(AccountInfo)" +
                  "could not be found in the lookup table: "
                  + "_Account_Info_L"
        End If
        Exit Function
ErrorHandler_Info_LookUp:
            tLog.WriteStubError Err.Number, Err.Source + ": "
                  + Err.Description
        Exit Function
RaiseError_Info_LookUp:
        Err.Raise Err.Number, Err.Source, Err.Description, Err.HelpFile,
                  Err.HelpContext
End Function
```

Summary

This class contains the following properties:

Property	Description
ExpectedException	Variant. A read-only value that represents the contents of the ExpectedException column in the current lookup table row.
ReturnValue	Variant. A read-only value that represents the contents of the ReturnValue column in the current lookup table row.
Value	Variant. A read-only value that represents the contents of the specified column ID or name in the current lookup table row.

This class contains the following methods:

Method	Description
Open	Opens the specified lookup table.
Search	Sets the cursor to the row in the current lookup table that contains the column value(s) passed to it.

LookUpTable.Open

Opens the specified lookup table.

Syntax

```
Open(TableName As String)
```

Element	Description
TableName	The name of the lookup table to open.

Comments

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

LookUpTable.Search

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

Syntax

```
Search(ParamNamesArray() As String,ParamValuesArray() As
   String) As Boolean
```

Element	Description
ParamNamesArray	An array containing one or more lookup-table column names.
ParamValuesArray	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.

Comments

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

If multiple rows contain the passed value(s), `FALSE` is returned.

TestLog Class

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

Overview

Extends `TSSLog`.

The items in this class are members of `RTCOMVPLib.TestLog`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this class.

TestLog Example

The following example logs a number of status messages about the various stages of a datapool operation.

```
Function ITestInterface_TestMain(ByVal args As Variant) As Variant

On Error GoTo MoveMoneyPerformScript_OnTestMainError

   'Declare variables.
   Dim lPrimeAccount As Long
   Dim lSecondAccount As Long
   Dim lAmount As Long
   Dim lTranType As Long
   Dim actRet As String
   Dim expRet As String
   Dim dp As New TSSDataPool
   Dim expErr As Variant
   Dim NumRows As Integer
   Dim Desc As String
   Dim Message As String

   'Set default values
   NumRows = 0

   'Initialize test data store and log services
   tms.LoginToTestRepository
   tms.StartTestServices "unittests\COM\RQACOMSample Ver 1.0
         (Rational QualityArchitect Sample for COM)\MoveMoneyPerform"

   'Create Object
   Dim obj As Object
   Set obj = CreateObject("RQACOMSample.MoveMoney")
```

```
'Open Datapool
dp.Open "_MoveMoney_Perform_D"

'Loop over datapool and perform test.
While dp.Fetch

   'Keep counter of number of rows fetched.
   NumRows = NumRows + 1

   'Get the column data from the datapool.
   lPrimeAccount = CLng(dp.Value("lPrimeAccount"))
   lSecondAccount = CLng(dp.Value("lSecondAccount"))
   lAmount = CLng(dp.Value("lAmount"))
   lTranType = CLng(dp.Value("lTranType"))
   expRet = CStr(dp.Value("expectedReturn"))
   expErr = dp.Value("expectedError")

   'Disable Default Error Handler
   On Error Resume Next

   'Clear the global error object in case it is set to something.
   Err.Clear

   'Call the method under test.
   actRet = obj.Perform(lPrimeAccount, lSecondAccount,
        lAmount, lTranType)

   'Save the Error info
   Dim actErr As Long
   Dim actErrDesc As String
   actErr = Err.Number
   actErrDesc = Err.Description

   'Enable Default Error Handler
   On Error GoTo MoveMoneyPerformScript_OnTestMainError

   'Evaluate the result of the method call
   If IsEmpty(expErr) Then

      If actErr <> 0 Then
         'Not expecting an error and one occurred.
         Desc = "Unexpected error" & "," & " " & CStr(actErr) &
              " was raised." & vbCrLf & actErrDesc
         Log.Message "Unexpected result", TSS_LOG_RESULT_FAIL, Desc
      Else
         'Not expecting an error and none occurred...
         'Therefore, check the expected expRet = actRet.

         If (expRet = actRet) Then
            Log.Message "Expected result", TSS_LOG_RESULT_PASS,
                 "Call to Perform returned expected value"
         Else
            Log.Message "Unexpected result", TSS_LOG_RESULT_FAIL,
               "Call to Perform returned unexpected value, "
```

```
                    & actRet & "."
               End If

            End If

         Else

            'expErr can represent Err.Number or Err.Description
            If actErr = expErr Or actErrDesc = expErr Then
               'Expecting an Error and the one raised matched
               'the expected error.
               Desc = "Expected error" & "," & " " & expErr
                    & " was raised."
               Log.Message "Expected Error", TSS_LOG_RESULT_PASS, Desc
            Else
               'Expecting an Error and either none was raised or...
               'the one raised was not the one we expected.
               Desc = "Expected error" & "," & " " & expErr &
                    " was not raised."
               Log.Message "Unexpected error", TSS_LOG_RESULT_FAIL, Desc
            End If

         End If

      Wend

      If NumRows = 0 Then
         'Datapool did not contain any rows.  Log a warning.
         Desc = "Datapool " & """ & "MoveMoneyPerform" & """
              & " is empty."
         Log.Message "Empty Datapool", TSS_LOG_RESULT_WARN, Desc
      End If

      'Execution to this point indicates success, so clear
      'any handled errors that may have occurred and continue.
      Err.Clear

      'Fall through to cleanup.
      'No Error will be logged because Err.Number now equals zero.

   MoveMoneyPerformScript_OnTestMainError:

      'If an error occurred, log it.
      If Err.Number <> 0 Then
         Message = "Unexpected error" & "," & " " & CStr(Err.Number) &
               " was raised."
         Log.Message Message, TSS_LOG_RESULT_FAIL, Err.Description
      End If

      'Close the datapool
      If Not dp Is Nothing Then
         dp.Close
         Set dp = Nothing
      End If
```

```
   'Shutdown test data store and log services
   tms.EndTestServices

End Function
```

Summary

This class contains the following methods:

Method	Description
`Message`	Logs a message.
`WriteError`	Logs an error that occurred during the execution of a test script.
`WriteStubError`	Logs information about an error that occurred during the execution of a Rational QualityArchitect stub.
`WriteStubMessage`	Logs 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 the section *Logging Class* on page 30.

TestLog.Message

Logs a message.

Syntax

`Message` (*`message`* `As String, [`*`result`* `As Integer], [`*`description`* `As String])`

Element	Description
`message`	Specifies the string to log.
`result`	Specifies the notification preference regarding the result of the call. Valid values: TSS_LOG_RESULT_NONE (default: no notification), TSS_LOG_RESULT_PASS, TSS_LOG_RESULT_FAIL, TSS_LOG_RESULT_WARN, TSS_LOG_RESULT_INFO. `0` specifies the default.

Element	Description
description	Specifies the string to be put in the entry's failure description field.

Example

For examples of this method, see *TestLog Example* on page 106.

TestLog.WriteError

Logs an error that occurred during the execution of a test script.

Syntax

```
WriteError(hr As Long, Description As String)
```

Element	Description
hr	The error to log.
description	A description of the error.

Comments

This method logs a Fail result for the test script.

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

TestLog.WriteStubError

Logs information about an error that occurred during the execution of a Rational QualityArchitect stub.

Syntax

```
WriteStubError(hr as Long, description As String)
```

Element	Description
hr	The error to log.
description	A description of the error.

Comments

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

Example

For an example of this method, see *LookUpTable Example* on page 103.

TestLog.WriteStubMessage

Logs a message relating to the execution of a Rational QualityArchitect stub, and also includes a description of the message.

Syntax

```
WriteStubMessage(bsMessage As String, Description As String)
```

Element	Description
bsMessage	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 103.

Verification Services 5

Introduction to Verification Points

This chapter provides the basic concepts involved in implementing verification points and 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 171.

About Verification Points

A *verification point* is 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 in either of the following ways:

- The interfaces described in *Database Verification Point Reference* on page 125. You typically use these are the interfaces when recording or writing scripts for testing COM/DCOM interaction with a database.

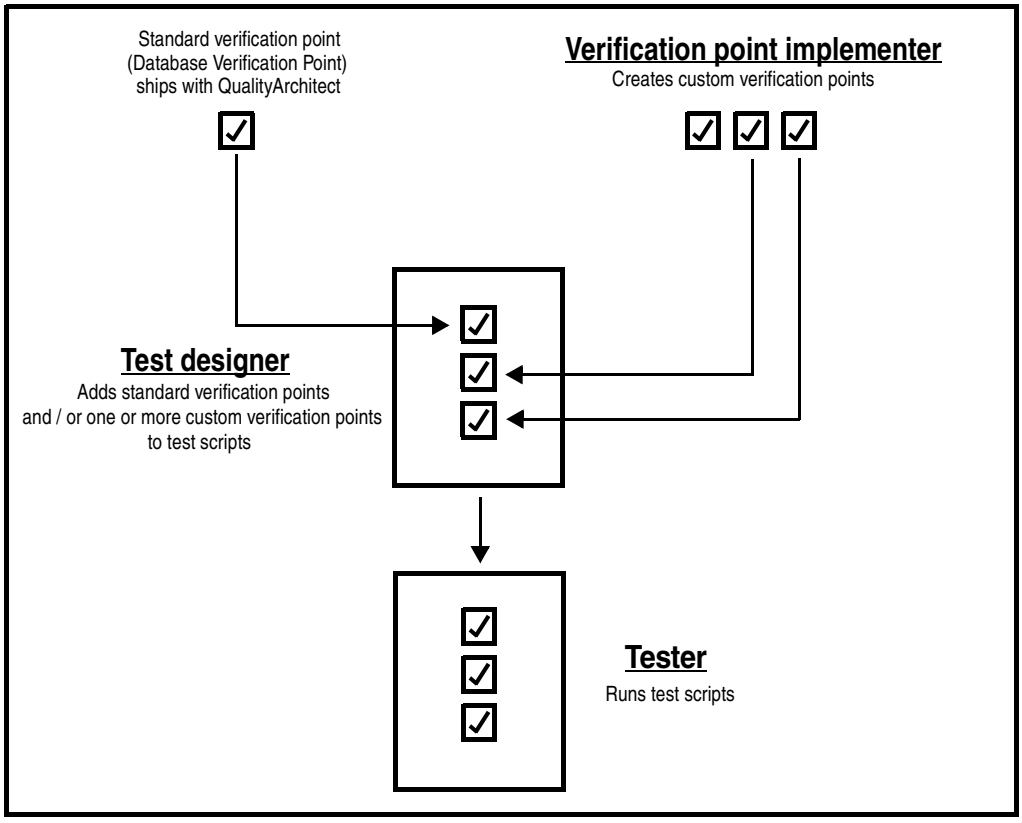
- To perform any other type of automated verification, implement a new verification point type. For example, if you want to verify the properties of an object, you must first implement classes that capture, encapsulate, and compare the object's properties. A verification point *implementer* implements verification point classes based on the abstract verification point interfaces described inChapter 7, *Verification Point Framework Reference*.

Roles in Working with Verification Points

The following testing team members use verification services. Depending upon the requirements of your site, the same person or different persons perform the different tasks.

- The verification point *implementer* implements new verification point types based on the verification point framework described in *Verification Point Framework Reference* on page 135.
- 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:



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.

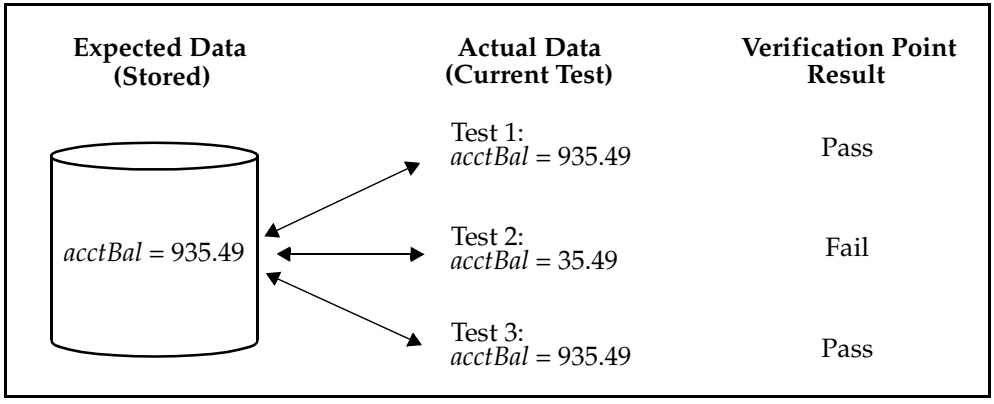
  Expected data can be data that is explicitly specified (for example, a person's name, social security number, or account number), or data that is the result of some calculation (for example, a monthly mortgage payment resulting from inputs of loan amount, interest rate, and number of payments).

- 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 `VPOPTION_EXPECT_FAILURE` option through the `Options` property of the `IVerificationPoint` interface, 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:

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

Static Verification Points

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

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

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

Dynamic Verification Points

With dynamic verification points, at test runtime, the expected data object is passed to the verification point. 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 any file.

When executing a dynamic verification point, 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.

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 a previous behavior of the component-under-test explicitly defining it.

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. The verification point framework does not provide expected and actual data objects. In contrast, with with static verification points, the framework provides both the expected and actual data objects) and, with dynamic verification points, 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 `IVerificationPointDataProvider` interface 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

You use the pre-defined database verification point for verifying data in a database. This is typically the verification point you use in writing scripts for COM/DCOM 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.

The verification point framework contains the following interfaces:

- `IVerificationPoint`
- `IVerificationPointData`
- `IVerificationPointDataProvider`
- `IVerificationPointDataRenderer`
- `IVerificationPointComparator`

- `IVPFramework`
- `IVPPlumbing`

For details about the framework, see Chapter 7, *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 `IVerificationPoint` interface.

  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 SQL statements for the 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 and XML).

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

  This class encapsulates and serializes a single snapshot of either expected or actual data. The `IVerificationPointDataProvider` class implements the `CaptureData` method to populate an instance of this class. Or, you can populate it manually in the test script — for example, by literal values or by values from a datapool. Each implementation of the `IVerificationPointData` 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 `IVerificationPointDataProvider` 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.

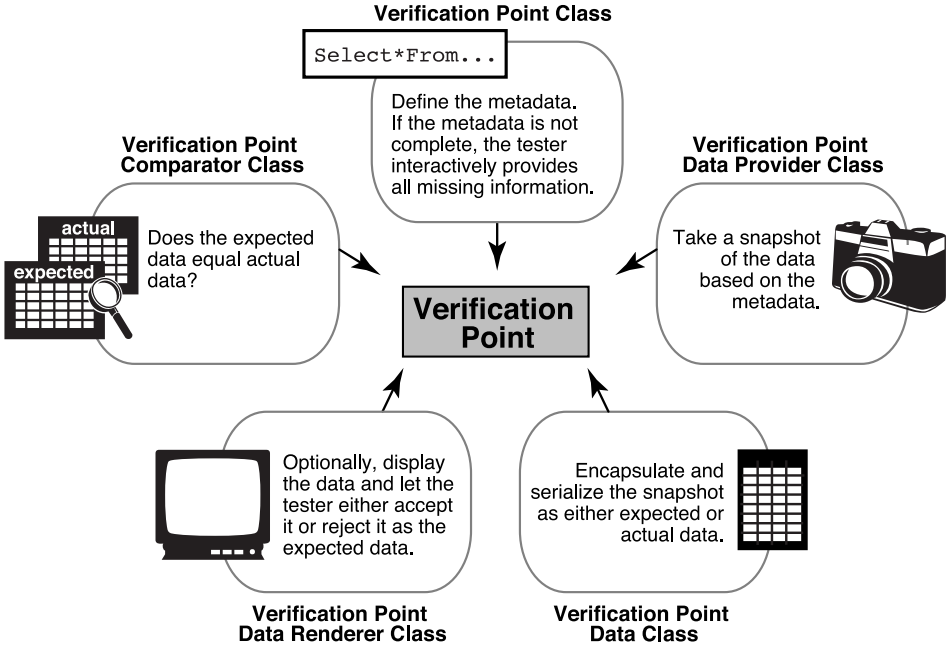
- A Verification Point Data Renderer class, which implements the framework's `IVerificationPointDataRenderer` interface.

  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 `VPOPTION_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 `IVerificationPointComparator` interface.

  This class provides a method to compare two `IVerificationPointData` 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:



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 actions outlined below 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 point.

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 `IVerificationPoint` 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. The following are two examples of this type of metadata:
  - With the pre-defined database verification point, the SQL statement that retrieves data from a database. (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 database).
- Possibly, one or more verification point options, such as whether to require case-sensitive matches of string data.

You can specify verification point metadata either explicitly or implicitly:

- Metadata that is specified *explicitly* in the test script is specified through user-defined `set...` methods in the specialized `IVerificationPoint` class.

Verification points that you generate through a Rational Rose model are defined explicitly — that is, the metadata is set through calls to the verification point's `set...` methods..

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 these 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 `DefineVP` method. This method runs a user-defined UI that prompts the tester for the metadata information. (The UI is 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 *IVerificationPoint Interface* on page 136.

Step 2. Execute the Verification Point

To execute a verification point, call the `PerformTest` method in the specialized `IVerificationPoint` 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 COM/DCOM object or a recordset in a database), pass `0` to the `PerformTest` method.

Using the metadata in the specialized `IVerificationPoint` 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 `VPOPTION_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 126.

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 `IVerificationPointData` interface.

After you create the expected data object, 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 *Dynamic Database Verification Point Example* on page 126.

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 `IVerificationPointData` interface.

After you create the expected and actual data objects, 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 data source. This is the verification point that you typically use in COM/DCOM 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.

Note: To see Interface Definition Language (IDL) equivalents of the methods in this chapter, see *IDL Equivalents* on page 205.

Requirements for Using the Database Verification Point

The database verification point requires Rational QualityArchitect.

Components of the Database Verification Point

The database verification point uses the following interfaces:

- *IDatabaseVP Interface* on page 127
- *IDatabaseVPData Interface* on page 128
- *IVerificationPoint Interface* on page 129
- *IVPFramework Interface* on page 131

Type Libraries

The interfaces in this chapter are defined in the type library RTIVP.TLB. Using the Visual Basic OLE/COM object viewer, you can find information about this type library under "Rational QualityArchitect COM Verification Point Interface Type Library" (RTIVP in the Object Browser).

The coclasses in this chapter are implemented in the type library RTCOMVP.DLL. Using the Visual Basic OLE/COM object viewer, you can find information about this type library under "Rational Quality Architect Playback Type Library" (RTCOMVPLib in the Object Browser).

RTIVP.TLB and RTCOMVP.DLL are located in the Rational\Rational Test\QualityArchitect folder.

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 callis the `PerformTest` method in the specialized Verification Point interface. 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 that are required to insert a verification point into a script, see *Setting Up Verification Points in Test Scripts* on page 121.

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.

```
Dim StaticVP As New DatabaseVP
StaticVP.VPname = "NewTest1"
StaticVP.SQL = "SELECT * FROM COFFEES"
StaticVP.ConnectionString =
        "Provider=MSDASQL.1;Persist Security Info=False;
        Data Source=COFFEEBREAK"
StaticVP.PerformTest 0
```

Dynamic Database Verification Point Example

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.

```
Dim DynamicVP As New DatabaseVP
Dim myExpected As New DatabaseVPData
Dim Columns(1 To 3) As String
Dim Row(1 To 3) As String
Dim result As VPResult

Columns(1) = "ID"
Columns(2) = "Brand"
Columns(3) = "Price"

Row(1) = "1"
```

```
Row(2) = "Peets"
Row(3) = "5.5"

myExpected.Columns = Columns
myExpected.Row(0) = Row

DynamicVP.VPname = "DynamicVP"
DynamicVP.SQL = "SELECT * FROM COFFEES WHERE ID = 1"
DynamicVP.ConnectionString =
        "Provider=MSDASQL.1;Persist Security Info=False;
        Data Source=COFFEEBREAK"
result = DynamicVP.PerformTest(0, myExpected)
```

IDatabaseVP Interface

Implements a database verification point.

The `DatabaseVP` object contains the database verification point name. It also contains options that affect the behavior of the verification point.

To execute the database verification point, call the `PerformTest` method. This method is inherited from the implemented `IVPFramework` interface.

Overview

Extends `IVerificationPoint`.

`IVerificationPoint` extends `IVPFramework`.

`IVPFramework` extends `IDispatch`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface contains the following properties:

Property	Description
ConnectionString	`String`. The information needed to connect to the data source.
SQL	`String`. The SQL statement to send to the data source.

In addition to these properties, you can also use the properties in the `IVerificationPoint` interface (page 129) and `IVPFramework` interface (page 131).

Note: This interface contains no methods of its own. However it does contain the `PerformTest` method, which is contained in `VPFramework`. Call `PerformTest` to run a database verification point. For information about `PerformTest`, see page 132.

IDatabaseVPData Interface

Using this interface, you can get and set properties relating to the rows and columns in the captured data object. The data object is encapsulated in table form.

Overview

Extends `IVerificationPointData`.

`IVerificationPointData` extends `IDispatch`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface contains the following properties:

Property	Description
`NumCols`	`Long`. The number of columns in the table.
`NumRows`	`Long`. The number of rows in the table.
`Columns`	`Variant`. An array of column names.
`Row`	`Variant`. An array of rows, each of which is an array of string values.

IVerificationPoint Interface

Provides methods and properties used for running a verification point.

Note: The only items documented in this section are those that you, the test script designer, need when manually adding or modifying a database verification point in a test script. Other properties and methods that a verification point implementer uses in this interface are not shown here. Implementers can find the complete interface in *IVerificationPoint Interface* on page 136.

Overview

Extends `IVPFramework`.

`IVPFramework` extends `IDispatch`.

Known subclass:
`IDatabaseVP`

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface includes the following property:

Property	Description
Options	Variant (implemented as Long). Options that affect the behavior of the verification point when capturing data, populating the data object, and comparing data objects. Options include the following pre-defined options plus any user-defined options: ▪ DATABASEOPTION_TRIM_WHITESPACE. Captured values should have whitespace trimmed from the right and left sides. ▪ The following options inherited from IVerificationPoint: ▫ VPOPTION_COMPARE_CASEINSENSITIVE. Text comparisons are not case-sensitive. By default, text comparisons are case sensitive. ▫ VPOPTION_EXPECT_FAILURE. The verification point's expected result is failure. If the comparison fails and this option is set, the verification point succeeds. ▫ VPOPTION_USER_ACKNOWLEDGE_BASELINE. 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.

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. Here are examples of turning options on and off:

- Turn on two options:

```
MyVP.Options = VPOPTION_COMPARE_CASEINSENSITIVE Or
VPOPTION_EXPECT_FAILURE
```

- Turn off the VPOPTION_EXPECT_FAILURE option, but leave all other options unchanged:

```
MyVP.Options = MyVP.Options And (Not VPOPTION_EXPECT_FAILURE)
```

IVPFramework Interface

Provides methods and properties used for running a database verification point.

Note: The only items documented in this section are those that you, the test script designer, need when manually adding or modifying a database verification point in a test script. Other properties and methods needed by a verification point implementer for this interface are not shown. Implementers can find the complete interface in *IVPFramework Interface* on page 145.

Overview

Extends `IDispatch`.

Known subclass:
`IVerificationPoint`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface includes the following property:

Property	Description
`VPName`	`String`. The name of the verification point.

This interface includes the following method:

Method	Description
`PerformTest`	Performs a static, dynamic, or manual verification point, depending upon the parameters that are passed to it.

VPFramework.PerformTest

Performs a static, dynamic, or manual verification point, depending upon the parameters that are passed to it.

Syntax

```
PerformTest(Object As Variant, [ExpectedData As Variant],
   [ActualData As Variant]) As Integer
```

Element	Description
Object	The component-under-test. If the verification point operates on a component that is not directly accessible (for example, a remote component or a database), the verification point object must contain the information needed to find the component-under-test, and the value of `Object` is ignored.
ExpectedData	An optional parameter that, if it exists, represents the 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.
ActualData	An optional parameter that, if it exists, represents the actual data object. This object was captured or constructed by code in the test script.

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.

Comments

The type of verification point that this method performs depends upon the parameters that you pass to it:

- *Object* only — static verification point.

  This type of verification point 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.

- *Object* and *ExpectedData* — dynamic verification point.

  This type of verification 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.

- *Object*, *ExpectedData*, and *ActualData* — manual verification point.

  This type of verification point allows a test script to capture or construct the actual data object, rather than relying on the framework to create the actual data object.

  A manual verification point simply compares the actual and expected data objects that are passed to it and logs the results of the comparison.

Verification Point Framework Reference 7

About the Verification Point Framework

The verification point *framework* is the underlying software that executes and manages a verification point. The framework serves two purposes:

- It provides the base 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.

Note: For guidance on using the methods in this chapter, see *Implementing a New Verification Point* on page 171.

Note: To see Interface Definition Language (IDL) equivalents of the methods in this chapter, see *IDL Equivalents* on page 205.

Requirements for Using the Verification Point Framework

The verification point framework requires Rational QualityArchitect.

Verification Point Framework Components

The framework contains the following interfaces:

- *IVerificationPoint Interface* on page 136
- *IVPFramework Interface* on page 145
- *IVerificationPointComparator Interface* on page 148
- *IVerificationPointData Interface* on page 149
- *IVerificationPointDataProvider Interface* on page 151
- *IVerificationPointDataRenderer Interface* on page 153
- *IVPPlumbing Interface* on page 155

Type Libraries

The interfaces in this chapter are defined in the type library RTIVP.TLB. Using the Visual Basic OLE/COM object viewer, you can find information about this type library under "Rational QualityArchitect COM Verification Point Interface Type Library" (RTIVP in the Object Browser).

The coclasses in this chapter are implemented in the type library RTCOMVP.DLL. Using the Visual Basic OLE/COM object viewer, you can find information about this type library under "Rational Quality Architect Playback Type Library" (RTCOMVPLib in the Object Browser).

IVerificationPoint Interface

An implementation of this interface must contain 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 connection string for connecting to a target database and the SQL statement for querying the database.

Don't confuse metadata with the data being verified. The data being verified is encapsulated by an implementation of the interface `IVerificationPointData`.

A verification point's metadata can be defined in either of these ways:

- Explicitly, through user-defined `set...` methods in your specialized `IVerificationPoint` interface.
- 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 `DefineVP` method in your specialized `IVerificationPoint` interface. 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 121.

An implementation of this interface 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 and XML).

Note: The current release only supports the .vpm and .ini formats.

Overview

Extends `IVPFramework`.

`IVPFramework` extends `IVPDispatch`.

Known subclass:
`IDatabaseVP`

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface contains the following property:

Property	Description
`Options`	Retrieves or sets options associated with the current verification point. This property stores any options that affect the behavior of `IVerificationPointComparator` or `IVerificationPointDataProvider`. Options include the following pre-defined options plus any user-defined options: ▪ `VPOPTION_COMPARE_CASEINSENSITIVE`. Text comparisons are not case sensitive. By default, text comparisons are case sensitive. ▪ `VPOPTION_EXPECT_FAILURE`. The verification point's expected result is failure. If the comparison fails and this option is set, the verification point succeeds. ▪ `VPOPTION_USER_ACKNOWLEDGE_BASELINE`. 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

This interface contains the following methods:

Method	Description
`CodeFactory GetConstructorInvocation`	Declares a variable for the verification point being constructed.
`CodeFactory GetExternalizedInputDecl`	Declares a variable for each value being input programmatically to the constructor.
`CodeFactory GetExternalizedInputInit`	Initializes a variable for each value being input programmatically to the constructor.
`CodeFactory GetNumExternalizedInputs`	Specifies the number of responses (inputs) that a tester provided when defining verification point metadata interactively through a UI.
`CodeFactory GetNumPropertySet`	Specifies the number of calls to `CodeFactoryGetPropertySet` that are required to fully specify the verification point's definition.
`CodeFactory GetPropertySet`	Sets a given property for the verification point.
`DefineVP`	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).

Note: For more information about these code factory methods, see *The Code Factory Methods* on page 176.

IVerificationPoint.CodeFactoryGetConstructorInvocation

Declares a variable for the verification point being constructed.

Syntax

```
CodeFactoryGetConstructorInvocation(Language As CTDScriptTypes)
   As String
```

Element	Description
Language	The language of your verification point implementation. Valid values include: ▪ CTD_SCRIPTTYPE_JAVA ▪ CTD_SCRIPTTYPE_VB ▪ CTD_SCRIPTTYPE_CPP

Return Value

A string containing a declaration for this verification point type. The declaration is syntactically correct for the specified language.

Comments

This method is never called from the test script. This method is called when a test script is generated from a Rose model.

This method provides a constructor call. This call plus the variables declared by `CodeFactoryGetExternalizedInputDecl` and set by `CodeFactoryGetPropertySet` allow the Rational QualityArchitect code generator to create a fully specified verification point in the generated test script code.

For information about the QualityArchitect code generator, see the *Generating Test Assets* in the *Rational QualityArchitect* online documentation.

IVerificationPoint.CodeFactoryGetExternalizedInputDecl

Declares a variable for each value being input programmatically to the constructor.

Syntax

```
CodeFactoryGetExternalizedInputDecl(Language As CTDScriptTypes,
   InputNumber As Integer) As String
```

Element	Description
Language	The language of your verification point implementation. Valid values include: ▪ CTD_SCRIPTTYPE_JAVA ▪ CTD_SCRIPTTYPE_VB ▪ CTD_SCRIPTTYPE_CPP
InputNumber	A number that indicates the current variable to declare. The number should start at 0 and increment by 1 in a loop.

Return Value

A line of code that declares the variable indicated by *InputNumber*. The code is syntactically correct for the specified language.

Comments

This method is never called from the test script. This method is called when a test script is generated from a Rose model.

The Rational Rose scenario test generator calls this method in a loop that iterates as many times as there are variables to declare (that is, the number retrieved from CodeFactoryGetNumExternalizedInputs).

Variables declared with this method are used in the code generated by CodeFactoryGetPropertySet and CodeFactoryGetExternalizedInputInit.

IVerificationPoint.CodeFactoryGetExternalizedInputInit

Initializes a variable for each value being input programmatically to the constructor.

Syntax

```
CodeFactoryGetExternalizedInputInit(Language As CTDScriptTypes,
   InputNumber As Integer) As String
```

Element	Description
Language	The language of your verification point implementation. Valid values include: ▪ CTD_SCRIPTTYPE_JAVA ▪ CTD_SCRIPTTYPE_VB ▪ CTD_SCRIPTTYPE_CPP
InputNumber	A number that indicates the current variable to initialize. The number should start at 0 and increment by 1 in a loop.

Return Value

A line of code that initializes the variable indicated by *InputNumber*. The code is syntactically correct for the specified language.

Comments

This method is never called from the test script. This method is called when a test script is generated from a Rose model.

The Rational Rose scenario test generator calls this method in a loop that iterates as many times as there are variables to initialize (that is, the number returned from CodeFactoryGetNumExternalizedInputs).

Variables initialized with this method are declared by CodeFactoryGetExternalizedInputDecl and used by CodeFactoryGetPropertySet.

IVerificationPoint.CodeFactoryGetNumExternalizedInputs

Specifies 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 `DefineVP` method.

Syntax

```
CodeFactoryGetNumExternalizedInputs(Language As CTDScriptTypes)
   As Integer
```

Element	Description
Language	The language of your verification point implementation. Valid values include: ▪ `CTD_SCRIPTTYPE_JAVA` ▪ `CTD_SCRIPTTYPE_VB` ▪ `CTD_SCRIPTTYPE_CPP`

Return Value

The number of tester inputs that require variable declarations to be made in the specified language.

Comments

This method is never called from the test script. This method is called when a test script is generated from a Rose model.

IVerificationPoint.CodeFactoryGetNumPropertySet

Specifies the number of `CodeFactoryGetPropertySet` calls that are required to fully specify the verification point's definition.

Syntax

```
CodeFactoryGetNumPropertySet(Language As CTDScriptTypes) As
   Integer
```

Element	Description
Language	The language of your verification point implementation. Valid values: ▪ `CTD_SCRIPTTYPE_JAVA` ▪ `CTD_SCRIPTTYPE_VB` ▪ `CTD_SCRIPTTYPE_CPP`

Return Value

The number of calls that are required to `CodeFactoryGetPropertySet`.

Comments

NumProps represents the total number of properties that need to be set for the verification point. Each property is set through a separate call to `CodeFactoryGetPropertySet` in syntax appropriate for the specified language.

This method is never called from the test script. This method is called when a test script is generated from a Rose model.

IVerificationPoint.CodeFactoryGetPropertySet

Sets a given property for the verification point.

Syntax

```
CodeFactoryGetPropertySet(Language As CTDScriptTypes,
   InputNumber As Integer) As String
```

Element	Description
Language	The language of your verification point implementation. Valid values: ▪ CTD_SCRIPTTYPE_JAVA ▪ CTD_SCRIPTTYPE_VB ▪ CTD_SCRIPTTYPE_CPP
InputNumber	A number that indicates the current property to set. The number should start at 0 and increment by 1 in a loop.

Return Value

A line of code that sets the property indicated by *InputNumber*. The code is syntactically correct for the specified language.

Comments

This method is never called from the test script. This method is called when a test script is generated from a Rose model.

The Rational Rose scenario test generator calls this method in a loop that iterates as many times as there are properties to set (that is, the number retrieved by CodeFactoryGetNumPropertySet).

IVerificationPoint.DefineVP

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

```
DefineVP()
```

Comments

The framework automatically invokes this method if the verification point is not fully defined when the `PerformTest` method is invoked.

When `DefineVP` is invoked, it should capture, presumably through some UI, any information necessary to fully define the metadata for the verification point, and then populate the verification point's attributes with the captured metadata. For example, the `DefineVP` method included with the database verification point provided with Rational QualityArchitect invokes the Query Builder software. Query Builder captures the connection string for the target database plus 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 interface uses the metadata to determine which data to capture.

If the verification point is being generated through a Rational Rose model, this method is invoked at script generation time. The resulting verification point metadata will automatically be provided to the test script. As a result, the `DefineVP` method will not be invoked at script playback time.

Implement this method only if you are implementing a new verification point.

IVPFramework Interface

Provides the method that a test designer uses to verify a component.

Overview

Extends `IDispatch`.

Known subclass:
`IVerificationPoint`

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface contains the following properties:

Property	Description
VPname	`String`. The name of the verification point.
VP	For internal use only.
Plumbing	For internal use only.
CodeFactorySuffix	`String`. A unique identifier to variable names assigned by the code factory methods. This unique suffix prevents name conflicts when multiple verification points are created in the same scope.

This interface contains the following method:

Method	Description
`PerformTest`	Performs a static, dynamic, or manual verification point, depending upon the parameters that are passed to it.

IVPFramework.PerformTest

Performs a static, dynamic, or manual verification point, depending upon the parameters that are passed to it.

Syntax

```
PerformTest(Object As Variant, [ExpectedData As Variant],
   [ActualData As Variant]) As Integer
```

Element	Description
Object	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. The `IVerificationPointDataProvider` interface passes this parameter to `CaptureData()` as its first parameter.

Element	Description
ExpectedData	An optional parameter which, if it exists, represents the 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.
ActualData	An optional parameter which, if it exists, represents the actual data object. This object was captured or constructed by code in the test script.

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.

Comments

The type of verification point that this method performs depends upon the parameters that you pass to it:

- *Object* only — static verification point.

  This type of verification point 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.

- *Object* and *ExpectedData* — dynamic verification point.

  This type of verification captures an actual data object from the object-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.

- *Object*, *ExpectedData*, and *ActualData* — manual verification point.

  This type of verification point allows a test script to capture or construct the actual data object, rather than relying on the framework to create the actual data object.

  A manual verification point simply compares the actual and expected data objects that are passed to it, and it logs the results of the comparison.

IVerificationPointComparator Interface

An interface implementing this interface provides a method that compares 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 interface is passed into the constructor of the abstract `VerificationPoint` interface and is used when that verification point needs to perform its comparison.

Overview

Extends `IDispatch`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

IVerificationPointComparator.Compare

Compares an expected data object with an actual data object and determines whether the test succeeds or fails.

```
compare(ExpectedData As IVerificationPointData, ActualData As
   IVerificationPointData, Options As Variant,
   FailureDescription As String) As Boolean
```

Element	Description
ExpectedData	The expected data object.
ActualData	The actual data object.

Element	Description
Options	Options that are passed from the Verification Point interface to qualify the comparison. Options can be pre-defined, such as VPOPTION_COMPARE_CASEINSENSITIVE, VPOPTION_EXPECT_FAILURE, and VPOPTION_USER_ACKNOWLEDGE_BASELINE, or any user-defined options.
FailureDescription	INPUT / OUTPUT. A value that contains the differences between the expected and actual data objects in a failed verification point. The failure description is written to the log. If you assign a value to this parameter, the method may change the value.

Return Value

A boolean value indicating whether the comparison passed or failed.

IVerificationPointData Interface

An interface implementing this interface encapsulates and serializes a single snapshot of either expected or actual data. A Verification Point Data Provider interface populates it through the `CaptureData` method , 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 `IVerificationPointData` interface must provide its own serialization methods. This enables support of all possible file formats. Use the `IPersistFile` interface to implement serialization for the encapsulated data.

Note: For the current Rational QualityArchitect release, Verification Point Data interfaces 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 interfaces 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 interface for use in dynamic and manual verification points.

Overview

Extends `IDispatch`.

Known subclass:
`IDatabaseVPData`

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

IVerificationPointData.FileExtension

Retrieves or specifies the extension of the file used to store the data object.

Syntax

```
FileExtension() As String
FileExtension(newVal As String)
```

Element	Description
newVal	A new file extension to use for storing data objects.

Return Value

The file extension currently used to store data objects.

Comments

The 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).

In the current release, .CSV is the only supported file format. Other formats will be supported in a future release.

IVerificationPointDataProvider Interface

An implementation of this interface creates a Verification Point Data object based on the verification point metadata in the specialized Verification Point object.

A component implementing this interface is a pluggable link between a Verification Point component (which defines a verification point's metadata) and a Verification Point Data component (which encapsulates and serializes the data for a verification point).

When you implement a Verification Point Data interface 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 interface knows about the structure of both the Verification Point Data interface (which it is building) and the Verification Point interface (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.

An implementation of this interface 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

Extends `IDispatch`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

IVerificationPointDataProvider.CaptureData

This method builds a `VerificationPointData` object.

Syntax

```
CaptureData(Object As Variant, VP As IVerificationPoint) As
   IVerificationPointData
```

Element	Description
Object	The object-under-test. The first parameter of the `PerformTest` method provides the contents of this parameter.
VP	The `IVerificationPoint` object that contains the verification point's metadata.

Return Value

An instance of the specialized `IVerificationPointData` interface populated with the captured data.

Comments

This method captures data according to the metadata in the I`VerificationPoint` interface. The framework may use the returned I`VerificationPointData` object as either an expected or an actual data object.

IVerificationPointDataRenderer Interface

An interface implementing this interface provides the capability of displaying the data stored in the Verification Point Data interface, allowing the tester to interactively accept or reject that data as the expected (baseline) data for a static verification point.

The test script uses the *Options* property to set the VPOPTION_USER_ACKNOWLEDGE_BASELINE option.

Overview

Extends IDispatch.

Known subclass:
DatabaseVPDataRenderer

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

IVerificationPointDataRenderer.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

```
DisplayAndValidateData(Data As IVerificationPointData) As
   Boolean
```

Element	Description
Data	INPUT / OUTPUT. The data to present to the tester for confirmation. The method may change this value after it is passed in.

Return Value

The return values is either:

- `true` if the tester accepts the displayed data
- `false` if the tester rejects the data.

Comments

This method is invoked by the verification point framework when the following conditions exist:

- The test script uses the `Options` property to set the `VPOPTION_USER_ACKNOWLEDGE_BASELINE` option.
- No expected data object exists in the datastore when the test script calls the `CaptureData` method of the Verification Point interface 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 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 no expected data is stored for the verification point. The next time the tester runs the script, the script again prompts the tester to accept the data.

IVPPlumbing Interface

Identifies the components of this verification point.

Overview

Extends `IDispatch`.

Applicability

Commonly used with Rational QualityArchitect.

Rational QualityArchitect is required for use of this interface.

Summary

This interface contains the following properties:

Property	Description
IsDefined	Boolean. If true, the verification point's metadata is fully specified. If false when a PerformTest method is invoked, the framework will call the DefineVP 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 property applies to the verification point metadata, not to the data itself that is captured in accordance with the metadata.
IsValid	Boolean. If true, indicates that the verification point was correctly instantiated, successfully captured, and is in a valid state — otherwise false.
VPComparator	String. The *progID* of the IVerificationPointComparator component for this verification point.
VPData	String. The *progID* of the IVerificationPointData component for this verification point.
VPDataProvider	String. The *progID* of the IVerificationPointDataProvider component for this verification point.
VPDataRenderer	String. The *progID* of the IVerificationPointDataRenderer component for this verification point.

This interface contains the following methods:

Method	Description
InitializeFramework	Specifies to the verification point framework the IDs of the components used by this verification point.
InitializeVP	For internal use only.

IVPPlumbing.InitializeFramework

Specifies to the verification point framework the IDs of the components used for this verification point.

Syntax

```
InitializeFramework(VPComparator As String, VPData As String,
   VPDataProvider As String, VPDataRenderer As String)
```

Element	Description
VPComparator	The *progID* of the IVerificationPointComparator component.
VPData	The *progID* of the IVerificationPointData component.
VPDataProvider	The *progID* of the IVerificationPointDataProvider component.
VPDataRenderer	The *progID* of the IVerificationPointDataRenderer component.

IVPPlumbing.InitializeVP

For internal use only.

Syntax

```
InitializeVP()
```

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

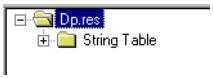
When you record a session, you indicate whether a script generated from the session will use datapools. A generated script that uses a datapool will include a block of code opening the datapool such as the following:

```
tssPool.Open LoadResString(testscript1), _
rtCOM.GetDatapoolAccessFlags, _
rtCOM.GetDatapoolOverrideList
```

The datapool name `testscript1` will be the same as the script name. Create and populate the actual datapool you want your test script to use, and replace `testscript1` with the datapool's name. Now, when your script plays back, it will retrieve configuration information from the project resource (.res) file regarding this datapool. So you need to edit the .res file as explained below.

Go to the project directory and double-click the project resource (.res) file. Note: if the file does not open when you double-click the .res file, this indicates that the Visual Basic resource editor was not installed on the local machine. In this case, install the resource editor and configure it to start up with Visual Basic.

When the resource file opens, you will see a display such as the following:



Click the + by the String Table folder to open it, and double-click on the String Table file inside. When the String Table opens, you will see a display such as the following:

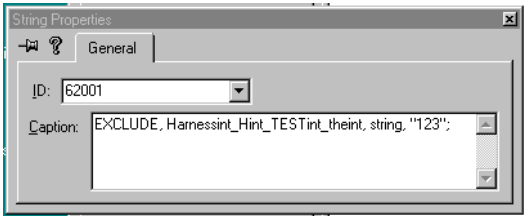
ID	Value	Caption
	61000	DP
	62000	DATAPOOL_CONFIG DP
	62001	EXCLUDE, Harnessint_Hint_TESTint_theint, string, "123";
	63000	62001

By default, on playback, the script will not use the datapool. This is because every field is set to `EXCLUDE`. So, for every field that should be populated from a datapool column of the same name, change `EXCLUDE` to `INCLUDE` by doing the following.

1 Click the line you want to change.

Inspect the string between `EXCLUDE` and the comma. It is a concatenation of these names: class, interface, method, and parameter. This string indicates an input field whose value may be supplied from a corresponding datapool column.

2 Right-click and select **Properties**: the String Properties dialog appears:



3 In the **Caption** panel, click `EXCLUDE` and change it to `INCLUDE`.

You can also change `EXCLUDE` to `OVERRIDE`. If you do this, the value in quotes at the end of the line will be inserted into the field on playback, rather than values from the corresponding datapool column.

4 End the String Properties dialog (click **x** in the upper right corner). The modified .res file is saved.

Now, when you run the script, the datapool will be used as indicated. To supply a different datapool configuration, re-edit the resource file and run the script again.

Finally, you may add access flags for the datapool: see *TSSDatapool.Open* on page 20 for a description of access flags. If you add access flags to the resource file, and do not specify any with the Open call, the access flags named in the resource file will be used for the datapool. Add access flags to the DATAPOOL_CONFIG line (62000), following the instructions above. For example, to specify the default datapool access flags, modify line 62000 as follows:

```
DATAPOOL_CONFIG DP DP_WRAP DP_SEQUENTIAL DP_SHARED
```

Note that, in the resource file, datapool access flags are specified without the `TSS_` prefix. They are otherwise identical to the access flags described for the Open method.

Synchronization Point and Shared Variable Configuration

Generated scripts that use synchronization points or shared variables will include ordinary synchronization point and shared variable method statements (see *Synchronization Class* on page 78). In addition, the .res files for those scripts will include String Table data. For example, suppose a generated script named IE5test includes these statements:

```
tssSync.SyncPoint "BlockUntilSaveComplete"
....
tssSync.SharedVarAssign "lineCounter",val,SHVOP_ADD
```

If you go to the project directory and open IE5test.res, you'll see String Table information similar to the following:

ID	Value	Caption
	61000	IE5test
	62000	DATAPOOL_CONFIG IE5test
	63000	62000
	64000	BlockUntilSaveComplete
	65000	lineCounter

For generated scripts such as in this example, no action is required. However, for any synchronization points or shared variables that you manually insert into a script, you must insure that the .res file contains information such as that shown above.

Adding String Table Data to a Resource File

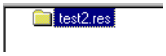
If you manually write a Visual Basic script that uses datapools, synchronization points, or shared variables, or if you manually add method statements for any of these to a generated script, you must add String Table data to the project resource file. The procedures are summarized below.

A generated Visual Basic project (.vbp) file contains the lines shown below. If you create a Visual Basic project from the IDE, you must add these reference lines to the project file.

```
Reference=*\G{00020430-0000-0000-C000-000000000046}#2.0#0#..\..\..\..\..
\WINNT\System32\stdole2.tlb#OLE Automation
Reference=*\G{175F8B42-FB70-11D3-99A4-00C04F5E9877}#1.0#0#..\..\..\..\..
\Program Files\Rational\Rational Test\rttsscom.dll#TestScriptServices
```

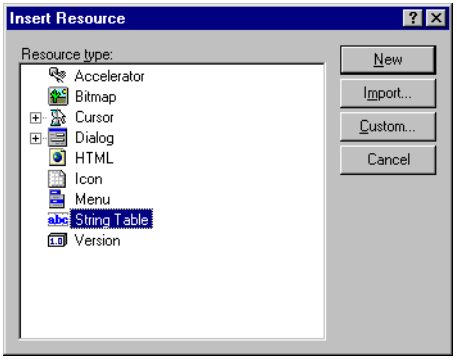
A Visual Basic script must be compiled into a .dll file. When you do this, a .rc file is produced. To get a .res file, you then compile the .rc file with the resource compiler.

When you open the .res file for a hand-written script (or for a generated script that included no datapool, synchronization point, and shared variable statements), you see an empty file.

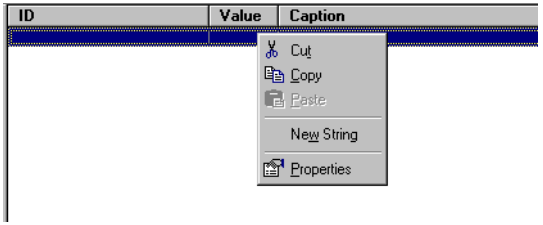


To add content to this file:

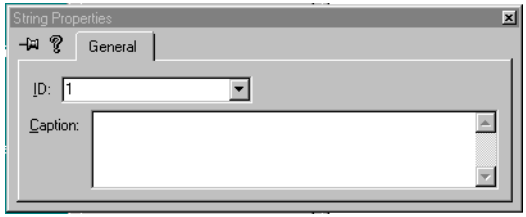
1 Select **Insert > Resource**. The Insert Resource dialog appears.



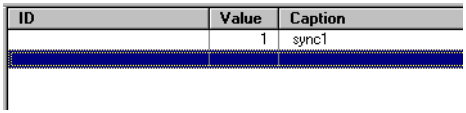
2 Select **String Table** and click **New**. An empty string table row (darkened) appears.



3 Place your cursor on the darkened row, click button 3, and select **Properties**. The String Properties dialog appears.



4 In the **ID** field, enter an ID number. In the **Caption** field, enter a value. For example, to configure a synchronization point named sync1, enter sync1. Then click the **x** on the upper right corner. This saves the String table entry as shown below.



Repeat the steps above until the resource file contains entries for all manually-inserted datapool, synchronization points, and shared variables.

RTCOM Support Class

B

About RTCOM

The Rational Test Component Object Model (RTCOM) class provides functions specific to the COM protocol. Visual Basic scripts that are generated from COM sessions use RTCOM class methods for error handling, datapool configuration, object monitoring, and logging operations.

Summary

In generated scripts, RTCOM methods are invoked via `rtcom`, a variable holding an object reference instantiated as follows:

```
Private RTCOM As New TestScriptServicesLib.RTCOMSupport
```

This is not shown in the examples.

The following table lists and describes the RTCOM member functions.

Function	Description
`ErrorArray`	Specifies the list of expected errors for the application under test.
`GetDatapoolAccessFlags`	Gets datapool access flags from the resource file.
`GetDatapoolOverrideList`	Gets datapool override column name/values from the resource file.
`Monitor`	Controls COM object monitoring.
`SetCMDID`	Sets the command ID for a COM method call.

ErrorArray

Specifies the list of expected error for the application under test.

Syntax

ErrorArray (*errList* as Long)

Element	Description
errList	An array of Longs specifying HRESULT values that should not be regarded as errors.

Error Codes

This method generates one of the following status codes:

- S_OK. Success.
- ERROR_TSS_ABORT. Abort in progress, probably resulting from a user request.
- ERROR_INVALID_PARM. A required argument is missing or invalid.
- ERROR_GET_TLS_INDEX. An internal storage error (unrecoverable) occurred.
- ERROR_OUT_OF_MEMORY. An attempt to allocate dynamic memory failed.

Comments

When a method completes, the COM interception mechanism within TSS needs to decide whether the method succeeded or failed. Without guidance, the mechanism will consider any HRESULT that has the error bit on to be a failure. It is possible that, for the application under test, certain HRESULTs that have the error bit set do not indicate failure. Use **ErrorArray** to pass any such HRESULTs to the interception mechanism so that it will not consider them failures for the application under test.

Example

This example passes three HRESULT values to the interception mechanism, so that it will not consider them to indicate failure.

```
Dim expectedErrs(2) As Long
expectedErrs(0) = 0
expectedErrs(1) = &H80040001
expectedErrs(2) = &H80040123
rtcom.ErrorArray expectedErrs
```

GetDatapoolAccessFlags

Gets datapool access flags from the resource file.

Syntax

```
GetDatapoolAccessFlags() As Long
```

Return Value

A 32-bit integer containing the access flags. If the resource file specifies multiple access flags, their bitmasks are merged in the integer.

Error Codes

This method generates one of the following status codes:

- `S_OK`. Success.
- `ERROR_TSS_ABORT`. Abort in progress, probably resulting from a user request.
- `ERROR_GET_TLS_INDEX`. An internal storage error (unrecoverable) occurred.
- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.

Comments

This call parses the resources associated with the script, extracts the datapool access flags, and returns them in a form suitable for passing to `TSSDatapool.Open`.

Example

This first example returns the datapool access flags in the resource file to `flags`. The second example invokes `TSSDatapool.Open`, whose arguments are passed by the `LoadResString` (a built-in function) from the resource file.

```
Dim Flags as Long
flags = rtcom.GetDatapoolAccessFlags

tssPool.Open LoadResString(TSSRES_datapoolname), _
             rtCOM.GetDatapoolAccessFlags, _
             rtCOM.GetDatapoolOverrideList
```

GetDatapoolOverrideList

Gets datapool override column name/values from the resource file.

Syntax

```
GetDatapoolOverrideList() as Variant
```

Return Values

A two-dimensional array of name/value pairs containing the datapool override list.

Error Codes

This method generates one of the following status codes:

- `S_OK`. Success.
- `ERROR_TSS_ABORT`. Abort in progress, probably resulting from a user request.
- `ERROR_GET_TLS_INDEX`. An internal storage error (unrecoverable) occurred.
- `ERROR_OUT_OF_MEMORY`. An attempt to allocate dynamic memory failed.

Comments

This call parses the resources associated with the script, extracts any datapool override name/value pairs, and returns them in a form suitable for passing to `TSSDatapool.Open`.

Example

This example invokes `TSSDatapool.Open`, whose arguments are passed by the `LoadResString` (a built-in function) from the resource file.

```
tssPool.Open LoadResString(TSSRES_datapoolname), _
             rtCOM.GetDatapoolAccessFlags, _
             rtCOM.GetDatapoolOverrideList
```

Monitor

Controls object monitoring and reporting.

Syntax

```
Monitor (toggle As String, [class As Variant], [method As
   Variant], [instance As Variant])
```

Element	Description
toggle	ON (monitor) or OFF (exclude from monitoring).
class	The name of a class include/exclude.
method	The name of a method in *class* to include/exclude.
instance	The instance of *class* to include/exclude.

Error Codes

This method generates one of the following status codes:

- S_OK. Success.
- ERROR_TSS_ABORT. Abort in progress, probably resulting from a user request.

Comments

By default, the COM interception mechanism monitors all non-TSS objects in order to keep track of them. This call allows you to exclude objects that should not be monitored. You can specify an object to exclude/include by specifying its class, method, instance, or a combination.

Example

This example excludes from monitoring all methods and instances of the class ADODB.

```
rtcom.Monitor "OFF", "ADODB"
```

SetCMDID

Sets the command ID for a COM method call.

Syntax

```
SetCMDID (cid As String)
```

Element	Description
cid	The command ID.

Error Codes

This method generates one of the following status codes:

- `S_OK`. Success.
- `ERROR_TSS_ABORT`. Abort in progress, probably resulting from a user request.

Comments

Command IDs appear in logs in order to improve their readability.

Example

This example sets the command ID for a method call to `test001`.

```
rtcom.SetCMDID "test001"
```

Implementing a New Verification Point

C

Introduction to Verification Point Implementation

The verification point framework is an open architecture. Using it, you can implement your own verification point types and execute them within the framework.

This appendix describes the steps necessary to implement a new verification point type. It has the following topics:

- *Fundamentals for Implementing a Verification Point* on page 172 describes the components you must implement.
- *Integrating Your Verification Point with QualityArchitect* on page 203 explains how your implemented components interact with the verification point framework and with the Rational QualityArchitect code generator to provide complete verification point services.

This appendix 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 appendix. This appendix assumes a sound working knowledge of COM/DCOM as well as an understanding of verification points.

In addition, this appendix assumes that you are implementing your new COM/DCOM verification point type in Visual C++. The examples and terminology in this appendix are targeted towards experienced C++ developers. If you want to implement a new verification point type in Visual Basic, you can use this appendix as a conceptual guide.

Note: To see Interface Definition Language (IDL) equivalents of the methods and properties you use to implement a new verification point type, see *IDL Equivalents* on page 205.

Fundamentals for Implementing a Verification Point

Rational QualityArchitect provides a framework for implementing COM/DCOM verification points. This framework includes interfaces for all of the components you must implement, as well as a `VPFramework` component that provides much of the required implementation for all verification points.

Any verification point type that you implement should inherit the `VPFramework` component's implementation. Since COM does not support implementation inheritance, you acccomplish this task through COM containment. With COM containment, methods and properties inherited from the framework appear as part of your verification point type. This minimizes the amount of code that a test designer has to write to perform a simple verification point, plus it eliminates the need for a QueryInterface operation.

For more information, see *Essential COM* by Don Box.

Task Summary

To implement a new verification point type, you must implement the interfaces and components documented in the following sections:

- *Interface for Your Verification Point Component* on page 172
- *The Verification Point Component* on page 173
- *Interface for Your Verification Point Data Component* on page 192
- *The Verification Point Data Component* on page 193
- *The Verification Point Data Comparator Component* on page 197
- *The Verification Point Data Provider Component* on page 200
- *The Verification Point Data Renderer Component* on page 202

Interface for Your Verification Point Component

Your verification point component's interface must contain properties or methods for defining the verification point. This interface must inherit from the `IVerificationPoint` interface — for example:

```
[
   object,
   uuid(7C4870B0-6E1A-11D4-9A26-0010A4E86989),
   dual,
   helpstring("IDatabaseVP Interface"),
   pointer_default(unique)
]
interface IDatabaseVP : IVerificationPoint
```

```
{
   [propget, helpstring("property ConnectionString")]
        HRESULT ConnectionString([out, retval] BSTR *pVal);
   [propput, helpstring("property ConnectionString")]
        HRESULT ConnectionString([in] BSTR newVal);
   [propget, helpstring("property SQL")] HRESULT SQL([out, retval]
        BSTR *pVal);
   [propput, helpstring("property SQL")] HRESULT SQL([in] BSTR newVal);
};
```

The Verification Point Component

Your specialized verification point must perform the following tasks:

- Define and maintain the metadata that describes the verification to be performed.
- Supply a UI that allows a tester to specify the metadata.
- Provide serialization services for the metadata.
- Serve as a "code factory" for the Rational QualityArchitect code generator. The code factory methods generate source code that can be inserted into a test script to create the instance of your verification point.

To enable your specialized Verification Point component to perform these tasks, you must implement the following interfaces:

- All of the methods in the `IVerificationPoint` interface (because your Verification Point interface inherits from this interface).
- Your Verification Point interface, which you defined in *Interface for Your Verification Point Component* on page 172.
- The `IPersistFile` interface.
- All of the methods in the `IVPFramework` interface (because the `IVerificationPoint` interface inherits from `IVPFramework`).

  Your `IVPFramework` methods should pass the calls through to a contained `VPFramework` object, thus inheriting the implementation through containment.

Here is an example class declaration for a Verification Point component:

```
class ATL_NO_VTABLE CDatabaseVP :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CDatabaseVP, &CLSID_DatabaseVP>,
   public IDispatchImpl<IDatabaseVP, &IID_IDatabaseVP,
        &LIBID_RTCOMVPLib>,
   public IPersistFile
{
public:
   CDatabaseVP()
   {
```

```
   }

DECLARE_REGISTRY_RESOURCEID(IDR_DATABASEVP)
DECLARE_PROTECT_FINAL_CONSTRUCT()

HRESULT FinalConstruct();

BEGIN_COM_MAP(CDatabaseVP)
   COM_INTERFACE_ENTRY(IDatabaseVP)
   COM_INTERFACE_ENTRY(IVerificationPoint)
   COM_INTERFACE_ENTRY2(IDatabaseVP, IDispatch)
   COM_INTERFACE_ENTRY(IPersistFile)
END_COM_MAP()

// IVerificationPoint
public:
   STDMETHOD(CodeFactoryGetConstructorInvocation)
            (CTDScriptTypes Language, BSTR * Code);
   STDMETHOD(CodeFactoryGetNumExternallizedInputs)
            (CTDScriptTypes Language, short * NumInputs);
   STDMETHOD(CodeFactoryGetExternalizedInputDecl)
            (CTDScriptTypes Language, short InputNumber, BSTR * Code);
   STDMETHOD(CodeFactoryGetExternalizedInputInit)(/*[in]*/
            CTDScriptTypes Language, /*[in]*/ short InputNumber,
            /*[out, retval]*/ BSTR *Code);
   STDMETHOD(CodeFactoryGetNumPropertySet)(/*[in]*/
            CTDScriptTypes Language, /*[out, retval]*/
            short *NumProps);
   STDMETHOD(CodeFactoryGetPropertySet)(/*[in]*/
            CTDScriptTypes Language, /*[in]*/ short InputNumber,
            /*[out, retval]*/ BSTR *Code);
   STDMETHOD(DefineVP)();

// IDatabaseVP
public:
   STDMETHOD(get_SQL)(/*[out, retval]*/ BSTR *pVal);
   STDMETHOD(put_SQL)(/*[in]*/ BSTR newVal);
   STDMETHOD(get_ConnectionString)(/*[out, retval]*/ BSTR *pVal);
   STDMETHOD(put_ConnectionString)(/*[in]*/ BSTR newVal);

// IPersistFile
public:
   STDMETHOD(GetCurFile)(LPOLESTR *ppszFileName);
   STDMETHOD(SaveCompleted)(LPCOLESTR pszFileName);
   STDMETHOD(Save)(LPCOLESTR pszFileName, BOOL fRemember);
   STDMETHOD(Load)(LPCOLESTR pszFileName, DWORD dwMode);
   STDMETHOD(IsDirty)();
   STDMETHOD(GetClassID)(CLSID *pClassID);

// IVPFramework
public:
// STDMETHOD(InitializeFramework)(BSTR Name, BSTR VPComparator,
         BSTR VPData, BSTR VPDataProvider, BSTR VPDataRenderer);
   STDMETHOD(PerformTest)(/*[in]*/ VARIANT Object, /*[optional, in]*/
```

```
         VARIANT ExpectedData, /*[optional, in]*/
         VARIANT ActualData, /*[out, retval]*/ enum VPResult *Result);
   STDMETHOD(get_VPname)(/*[out, retval]*/ BSTR *pVal);
   STDMETHOD(put_VPname)(/*[in]*/ BSTR newVal);
   STDMETHOD(get_Options)(/*[out, retval]*/ VARIANT *pVal);
   STDMETHOD(put_Options)(/*[in]*/ VARIANT newVal);
   STDMETHOD(get_VP)( IDispatch ** ppVP );
   STDMETHOD(put_VP)( IDispatch * pVP );
   STDMETHOD(get_Plumbing)( IDispatch ** ppPlumbing );
   STDMETHOD(put_Plumbing)( IDispatch * pPlumbing );
   STDMETHOD(get_CodeFactorySuffix)( BSTR* pVal );
   STDMETHOD(put_CodeFactorySuffix)( BSTR newVal );

private:
   HRESULT CalcIsDefined();
   CComPtr<IVPFramework> m_pFramework;
   LONG lOptions;
   _bstr_t bstConnectionString;
   _bstr_t bstSQL;
};
```

Implementing the IVerificationPoint Interface

This section describes the implementation of the following parts of the `IVerificationPoint` interface:

- The `DefineVP()` method
- The code factory methods
- The *`Options`* property

The DefineVP() Method

The `DefineVP()` method in the `IVerificationPoint` interface invokes a UI to capture the metadata definition of the verification point. When the tester dismisses the UI, this method should populate the verification point object with the metadata it captured.

The method should return `S_OK` if the metadata was successfully captured, and `E_VP_UNDEFINED` if it was not — for example:

```
STDMETHODIMP CDatabaseVP::DefineVP()
{
   ...

   // Invoke some GUI to capture the VP's definition, in this case,
   // the QueryBuilder.

   ...

   pQB->DoModal();
```

```
   pQB->get_Accepted( &vAccepted );
   if ( vAccepted.vt == VT_BOOL && vAccepted.boolVal == VARIANT_TRUE )
   {
      pQB->get_Connection( &vConnection );
      pQB->get_SQL( &vSQL );
      if ( vConnection.vt == VT_BSTR && vSQL.vt == VT_BSTR )
      {
         bstConnectionString = vConnection.bstrVal;
         bstSQL = vSQL.bstrVal;
      }
      else
      {
         bDefined = false;
      }
      if ( *((LPCWSTR)bstConnectionString) == L'\0'  ||
           *((LPCWSTR)bstSQL) == L'\0' )
      {
         bDefined = false;
      }
   }
   else // User canceled, or internal error in Query Builder component.
   {
      bDefined = false;
   }

   if ( bDefined == false )
      return E_VP_UNDEFINED;
   else
      return S_OK;

}
```

The Code Factory Methods

The *code factory* methods generate source code that is capable of creating instances of your verification point type.

The code factory methods are similar in function to ActiveX controls that provide additional design-time behavior that integrates with the Visual C++ or Visual Basic environments.

The Rational QualityArchitect code generator uses the code factory methods to insert verification points into generated test scripts. If a tester using QualityArchitect wants to insert one of your verification points into a generated test script, the QualityArchitect code generator creates an instance of your verification point, and then calls the `DefineVP()` method to present the tester with the UI you have created. Once the tester supplies the metadata through your UI, the code generator invokes the code factory methods to return source code. When the returned source code is inserted into the test script, a verification point is created from the metadata that the tester supplied.

For information about how the QualityArchitect code generator uses the code factory methods, see *Integrating Your Verification Point with QualityArchitect* on page 203.

Following are the code factory methods that you implement:

- `CodeFactoryGetConstructorInvocation()` returns a line of code that invokes your verification point's constructor. The code is syntactically correct for the given language.
- `CodeFactoryGetNumExternalizedInputs()` returns the number of externalized input variables required for code generation of this verification point.
- `CodeFactoryGetExternalizedInputDecl()` returns a line of code that declares the specified externalized input for this verification point. The code is syntactically correct for the given language.
- `CodeFactoryGetExternalizedInputInit()` returns a line of code that initializes the specified externalized input for this verification point. The code is syntactically correct for the given language.
- `CodeFactoryGetNumPropertySet( )` returns the number of property-set calls required to fully specify the metadata for this verification point.
- `CodeFactoryGetPropertySet()` returns a line of code that sets the specified property for this verification point. The code is syntactically correct for the given language.

There are also *`CodeFactorySuffix`* property methods. The framework implements these methods — you only pass them through. You use the suffix when constructing the externalized variables that are returned by the following methods:

- `CodeFactoryGetPropertySet()`
- `CodeFactoryGetExternalizedInputDecl()`
- `CodeFactoryGetExternalizedInputInit()`

The suffix ensures that externalized variable names from multiple verification points in the same scope are unique. If the Rational QualityArchitect code generator sets the suffix, append the suffix to each externalized variable that is declared, initialized, and set by these methods. This allows the Rational Quality Architect code generator to insert more than one verification point into a test script without risk of variable name conflicts.

Example of Code Factory Methods

The following code listing illustrates the use of the code factory methods:

```
STDMETHODIMP CDatabaseVP::get_CodeFactorySuffix(BSTR *pVal)
{
   return m_pFramework->get_CodeFactorySuffix(pVal);
}

STDMETHODIMP CDatabaseVP::put_CodeFactorySuffix(BSTR newVal)
{
   return m_pFramework->put_CodeFactorySuffix(newVal);
}


STDMETHODIMP
CDatabaseVP::CodeFactoryGetConstructorInvocation(CTDScriptTypes
Language, BSTR * Code)
{
   if (Code == NULL)
      return E_POINTER;

   // Create a line of code which constructs this type of VP.
   _bstr_t bsCode;
   BSTR bsName = NULL;
   BSTR bsSuffix = NULL;

   get_VPname(&bsName);
   get_CodeFactorySuffix(&bsSuffix);

   if ( bsName == NULL )
   {
      return E_INVALIDARG;
   }

   switch ( Language )
   {
   case CTD_SCRIPTTYPE_VB: // VB
      bsCode = L"Dim ";
      bsCode += bsName;
      if ( bsSuffix ) bsCode += bsSuffix;
      bsCode += L" As New DatabaseVP";
      break;

   default:
      return E_INVALIDARG;
   }

   *Code = bsCode.copy();

   return S_OK;
}

STDMETHODIMP
```

```
CDatabaseVP::CodeFactoryGetNumExternallizedInputs(CTDScriptTypes
     Language, short * NumInputs)
{
   if (NumInputs == NULL)
      return E_POINTER;

   // Only VB is supported currently
   if (Language != CTD_SCRIPTTYPE_VB)
      return E_INVALIDARG;

   if ( lOptions == 0 )
      *NumInputs = 3;
   else
      *NumInputs = 4;

   return S_OK;
}

STDMETHODIMP
CDatabaseVP::CodeFactoryGetExternalizedInputDecl(CTDScriptTypes
     Language, short InputNumber, BSTR * Code)
{
   if (Code == NULL)
      return E_POINTER;

   // Only VB is supported currently
   if (Language != CTD_SCRIPTTYPE_VB)
      return E_INVALIDARG;

   _bstr_t bsCode;
   BSTR bsSuffix = NULL;

   get_CodeFactorySuffix(&bsSuffix);

   switch ( InputNumber )
   {
   case 1: // VPName
      bsCode = L"Dim VPname";
      if ( bsSuffix ) bsCode += bsSuffix;
      bsCode += L" As String";
      break;

   case 2: // Connection String
      bsCode = L"Dim VPConnectString";
      if ( bsSuffix ) bsCode += bsSuffix;
      bsCode += L" As String";
      break;

   case 3: // SQL Statement
      bsCode = L"Dim VPSQL";
      if ( bsSuffix ) bsCode += bsSuffix;
      bsCode += L" As String";
      break;
```

```
      case 4: // Options
         bsCode = L"Dim VPOptions";
         if ( bsSuffix ) bsCode += bsSuffix;
         bsCode += L" As Integer";
         break;

      default:

         break;
      }

      *Code = bsCode.copy();

      return S_OK;
   }

   STDMETHODIMP CDatabaseVP::CodeFactoryGetExternalizedInputInit(/*[in]*/
         CTDScriptTypes Language, /*[in]*/ short InputNumber,
         /*[out,retval]*/ BSTR *Code)
   {
      if (Code == NULL)
         return E_POINTER;

      // Only VB is supported currently
      if (Language != CTD_SCRIPTTYPE_VB)
         return E_INVALIDARG;

      _bstr_t bsCode;
      BSTR bsName = NULL;
      BSTR bsSuffix = NULL;

      get_VPname(&bsName);
      get_CodeFactorySuffix(&bsSuffix);

      if ( bsName == NULL )
      {
         // TODO: Enum this error condition!
         return E_INVALIDARG;
      }

      switch ( InputNumber )
      {
      case 1: // VPname
         bsCode = L"VPname";
         if ( bsSuffix ) bsCode += bsSuffix;
         bsCode += L" = \"";
         bsCode += bsName;
         bsCode += L"\"";
         break;

      case 2: // Connection String
         bsCode = L"VPConnectString";
         if ( bsSuffix ) bsCode += bsSuffix;
         bsCode += L"= \"";
```

```
      bsCode += bstConnectionString;
      bsCode += L"\"";
      break;

   case 3: // SQL Statement
      bsCode = L"VPSQL";
      if ( bsSuffix ) bsCode += bsSuffix;
      bsCode += L" = \"";
      bsCode += bstSQL;
      bsCode += L"\"";
      break;

   case 4: // Options
      bsCode = L"VPOptions";
      if ( bsSuffix ) bsCode += bsSuffix;
      bsCode += L" = ";
      bsCode += L"VPOptions";
      break;

   default:

      break;
   }

   *Code = bsCode.copy();

   return S_OK;

}


STDMETHODIMP CDatabaseVP::CodeFactoryGetNumPropertySet(/*[in]*/
     CTDScriptTypes Language, /*[out, retval]*/ short *NumProps)
{
return CodeFactoryGetNumExternallizedInputs(Language, NumProps);
}


STDMETHODIMP CDatabaseVP::CodeFactoryGetPropertySet(/*[in]*/
     CTDScriptTypes Language, /*[in]*/ short InputNumber,
     /*[out, retval]*/ BSTR *Code)
{
   if (Code == NULL)
      return E_POINTER;

   // Only VB is supported currently
   if (Language != CTD_SCRIPTTYPE_VB)
      return E_INVALIDARG;

   _bstr_t bsCode;
   BSTR bsName = NULL;
   BSTR bsSuffix = NULL;

   get_VPname(&bsName);
```

```
    get_CodeFactorySuffix(&bsSuffix);

    if ( bsName == NULL )
    {
       // TODO: Enum this error condition!
       return E_INVALIDARG;
    }

    switch ( InputNumber )
    {
    case 1: // VPname
       bsCode = bsName;
       if ( bsSuffix ) bsCode += bsSuffix;
       bsCode += L".VPname = ";
       bsCode += L"VPname";
       if ( bsSuffix ) bsCode += bsSuffix;
       break;

    case 2: // Connection String
       bsCode = bsName;
       if ( bsSuffix ) bsCode += bsSuffix;
       bsCode += L".ConnectionString = ";
       bsCode += L"VPConnectString";
       if ( bsSuffix ) bsCode += bsSuffix;
       break;

    case 3: // SQL Statement
       bsCode = bsName;
       if ( bsSuffix ) bsCode += bsSuffix;
       bsCode += L".SQL = ";
       bsCode += L"VPSQL";
       if ( bsSuffix ) bsCode += bsSuffix;
       break;

    case 4: // Options
       bsCode = bsName;
       if ( bsSuffix ) bsCode += bsSuffix;
       bsCode += L".Options = ";
       bsCode += L"VPOptions";
       if ( bsSuffix ) bsCode += bsSuffix;
       break;

    default:

       break;
    }

    *Code = bsCode.copy();

    return S_OK;

}
```

Sample Code Factory Output

The following example illustrates the output produced by calling the code factory methods for a database verification point on a fully specified database verification point object.

In this example, the caller performed these preliminary steps:

- Created a `DatabaseVP` object and set its *`VPname`* property to `Simple`.
- Set its code factory suffix to `1`.
- Called the `DefineVP()` method.

This is consistent with the behavior you see with the Rational QualityArchitect code generator. In the QueryBuilder invoked by the `DefineVP()` method, the caller specified an ODBC data source called `COFFEEBREAK`, and built the SQL statement `"SELECT * FROM COFFEES"`.

The following samples show what the code factory methods return when they are invoked for this database verification point object:

- Returned from `CodeFactoryGetConstructorInvocation()`:

```
Dim Simple1 As New DatabaseVP
```

- Returned from `CodeFactoryGetNumExternalizedInputs()`:

```
3
```

- Returned from `CodeFactoryGetExternalizedInputDecl()`:

```
Dim VPname1 As String
Dim VPConnectString1 As String
Dim VPSQL1 As String
```

- Returned from `CodeFactoryGetExternalizedInputInit()`:

```
VPname1 = "Simple"
VPConnectString1= "Provider=MSDASQL.1;Persist Security Info=False;
        Data Source=COFFEEBREAK"
VPSQL1 = "select * from coffees"
```

- Returned from `CodeFactoryGetPropertySet()`:

```
Simple1.VPname = VPname1
Simple1.ConnectionString = VPConnectString1
Simple1.SQL = VPSQL1
```

The Options Property

Several flags are pre-defined for all verification points (see the property table in *Summary* on page 137), but you can add additional flags for your new verification point type. If you do so, use a single bit to represent each option, with the first available bit being the fourth bit (`0x8`).

The *`Options`* property is defined in the `IVerificationPoint` interface as a `Variant`, but it is implemented as a `Long` bitfield.

The following example shows a mechanism for creating new options:

```
STDMETHODIMP CDatabaseVP::get_Options(/*[out, retval]*/ VARIANT *pVal)
{
   pVal->vt = VT_I4;
   pVal->lVal = lOptions;
   return S_OK;

}

STDMETHODIMP CDatabaseVP::put_Options(/*[in]*/ VARIANT newVal)
{
   switch (newVal.vt)
   {
   case VT_I4:
      lOptions = newVal.lVal;
      break;
   case VT_I2:
      lOptions = (LONG) newVal.iVal;
      break;
   case VT_UI4:
      lOptions = (LONG) newVal.ulVal;
      break;
   case VT_UI2:
      lOptions = (LONG) newVal.uiVal;
      break;
   case VT_UINT:
      lOptions = (LONG) newVal.uintVal;
      break;
   case VT_INT:
      lOptions = (LONG) newVal.intVal;
      break;

   default:
      return S_FALSE;
   }

   return S_OK;
}
```

Implementing the Methods in Your Verification Point Interface

Your verification point component's interface contains properties or methods for defining the verification point's metadata— for example:

```
STDMETHODIMP CDatabaseVP::get_ConnectionString(BSTR *pVal)
{
   if (pVal == NULL)
      return E_POINTER;
   *pVal = SysAllocString(bstConnectionString);
   return S_OK;
}

STDMETHODIMP CDatabaseVP::put_ConnectionString(BSTR newVal)
{
   bstConnectionString = newVal;
   CalcIsDefined();
   return S_OK;
}

STDMETHODIMP CDatabaseVP::get_SQL(BSTR *pVal)
{
   if (pVal == NULL)
      return E_POINTER;
   *pVal = SysAllocString(bstSQL);
   return S_OK;
}

STDMETHODIMP CDatabaseVP::put_SQL(BSTR newVal)
{
   bstSQL = newVal;
   CalcIsDefined();
   return S_OK;
}
```

Implementing the IPersistFile Interface

The framework uses the following two `IPersistFile` interface methods to serialize your verification point's metadata. The other `IPersistFile` methods can simply return `E_NOTIMPL`:

- `Load()`. loads your verification point's metadata from a verification point metafile (.vpm file). The framework calls this method if a test script calls the `PerformTest()` method when the verification point is not yet fully defined (that is, when one or more required pieces of metadata are missing).
- `Save()`. saves your verification point's metadata to a .vpm metafile. The framework calls this method when both of the following conditions exist:
  - No metafile currently exists.

- The `PerformTest()` method is called when the verification point is not yet fully defined.

When these conditions exist, the framework first calls the `DefineVP()` method to prompt the tester for the metadata, and then calls `Save()` to store the metadata for future runs of this verification point.

The framework also calls `Save()` to write a copy of the metafile to the Log folder for use by the Grid Comparator.

The following example illustrates metadata serialization:

```
STDMETHODIMP CDatabaseVP::GetClassID(CLSID *pClassID)
{
   return E_NOIMPL;
}

STDMETHODIMP CDatabaseVP::IsDirty()
{
   return E_NOIMPL;
}

STDMETHODIMP CDatabaseVP::SaveCompleted(LPCOLESTR pszFileName)
{
   return E_NOIMPL;
}

STDMETHODIMP CDatabaseVP::GetCurFile(LPOLESTR *ppszFileName)
{
   return E_NOIMPL;
}

STDMETHODIMP CDatabaseVP::Load(LPCOLESTR pszFileName, DWORD dwMode)
{
   TCHAR szBuffer[4096];
   _bstr_t bstFile(pszFileName);
   long lReadOptions = 0;

   GetPrivateProfileString(_T("Definition"), _T("Case ID"), _T("\n"),
        szBuffer, 4096, (LPCTSTR) bstFile );

   // Verify that the file exists and that there is at least a VP name.
   if ( _tcscmp(szBuffer, _T("\n")) == 0 )
      return E_VP_FILENOTFOUND;
   else
      put_VPname(_bstr_t(szBuffer));

   GetPrivateProfileString(_T("Definition"),
        _T(  "Verification Method"), _T("\n"),  szBuffer, 4096,
        (LPCTSTR) bstFile );

   // Check to see if comparison is Case Insensitive
   if ( _tcscmp(szBuffer, _T("CaseInsensitive")) == 0 )
```

```
      lReadOptions |= VPOPTION_COMPARE_CASEINSENSITIVE;

   GetPrivateProfileString(_T("Definition"), _T("Expected Result"),
        _T("\n"),  szBuffer, 4096, (LPCTSTR) bstFile );

   // Check to see if expected result is failure
   if ( _tcscmp(szBuffer, _T("Failure")) == 0 )
      lReadOptions |= VPOPTION_EXPECT_FAILURE;

   GetPrivateProfileString(_T("DatabaseVP"), _T("Connection String"),
        _T("\n"),  szBuffer, 4096, (LPCTSTR) bstFile );

   // Connection String is a required field.
   if ( _tcscmp(szBuffer, _T("\n")) == 0 )
      return E_VP_BADFILE;
   else
      put_ConnectionString(_bstr_t(szBuffer));

   GetPrivateProfileString(_T("DatabaseVP"), _T("SQL"), _T("\n"),
        szBuffer, 4096, (LPCTSTR) bstFile );

   // SQL is a required field.
   if ( _tcscmp(szBuffer, _T("\n")) == 0 )
      return E_VP_BADFILE;
   else
      put_SQL(_bstr_t(szBuffer));

   GetPrivateProfileString(_T("DatabaseVP"), _T("Trim Whitespace"),
        _T("\n"),  szBuffer, 4096, (LPCTSTR) bstFile );

   // Check to see if we should trim whitespace
   if ( _tcscmp(szBuffer, _T("1")) == 0 )
      lReadOptions |= DATABASEVPOPTION_TRIM_WHITESPACE;

   lOptions = lReadOptions;

   CalcIsDefined();

   return S_OK;
}

STDMETHODIMP CDatabaseVP::Save(LPCOLESTR pszFileName, BOOL fRemember)
{
   BSTR bsName = NULL;

   get_VPname( &bsName );
   _bstr_t bstName(bsName);
   _bstr_t bstFile(pszFileName);

   // Write [Definition] section
   WritePrivateProfileString(_T("Definition"), NULL, NULL,
        (LPCTSTR)bstFile);
   WritePrivateProfileString(_T("Definition"), _T("Case ID"),
        (LPCTSTR)bstName, (LPCTSTR)bstFile);
```

```
   WritePrivateProfileString(_T("Definition"), _T("Type"),
         _T("Object Data"), (LPCTSTR)bstFile);
   WritePrivateProfileString(_T("Definition"), _T("Data Test"),
         _T("Contents"), (LPCTSTR)bstFile);

   if ( lOptions & VPOPTION_COMPARE_CASEINSENSITIVE )
      WritePrivateProfileString(_T("Definition"),
         _T("Verification Method"), _T("CaseInsensitive"),
         (LPCTSTR)bstFile);
   else
      WritePrivateProfileString(_T("Definition"),
         _T("Verification Method"), _T("CaseSensitive"),
         (LPCTSTR)bstFile);

   if ( lOptions & VPOPTION_EXPECT_FAILURE )
      WritePrivateProfileString(_T("Definition"),
         _T("Expected Result"), _T("Failure"), (LPCTSTR)bstFile);

   // Write [DatabaseVP] section
   WritePrivateProfileString(_T("DatabaseVP"), NULL, NULL,
         (LPCTSTR)bstFile);
   WritePrivateProfileString(_T("DatabaseVP"),
         _T("Connection String"), (LPCTSTR)bstConnectionString,
         (LPCTSTR)bstFile);
   WritePrivateProfileString(_T("DatabaseVP"), _T("SQL"),
         (LPCTSTR)bstSQL, (LPCTSTR)bstFile);

   if ( lOptions & DATABASEVPOPTION_TRIM_WHITESPACE )
      WritePrivateProfileString(_T("DatabaseVP"),
         _T("Trim Whitespace"), _T("1"), (LPCTSTR)bstFile);

   return S_OK;
}
```

Implementing the IVPFramework Interface

`IVPFramework` is the base class for `IVerificationPoint`, and `IVerificationPoint` is the base class for your verification point's interface. Consequently, you must provide entry points for all of the methods in the `IVPFramework` interface.

Since the implementation of the `IVPFramework` methods is provided in the `VPFramework` component, your class must simply construct a `VPFramework` object, and then pass each of the `IVPFramework` methods to that `VPFramework` object.

This form of implementation inheritance is called *containment*, and it is illustrated in the following example:

```
STDMETHODIMP CDatabaseVP::get_VPname(BSTR *pVal)
{
   return m_pFramework->get_VPname(pVal);
}
```

```
STDMETHODIMP CDatabaseVP::put_VPname(BSTR newVal)
{
   return m_pFramework->put_VPname(newVal);
}

STDMETHODIMP CDatabaseVP::get_VP( IDispatch ** ppVP )
{
   return m_pFramework->get_VP(ppVP);
}

STDMETHODIMP CDatabaseVP::put_VP( IDispatch * pVP )
{
   return m_pFramework->put_VP(pVP);
}

STDMETHODIMP CDatabaseVP::get_Plumbing( IDispatch ** ppPlumbing )
{
   return m_pFramework->get_Plumbing(ppPlumbing);
}

STDMETHODIMP CDatabaseVP::put_Plumbing( IDispatch * pPlumbing )
{
   return m_pFramework->put_Plumbing(pPlumbing);
}

STDMETHODIMP CDatabaseVP::get_CodeFactorySuffix(BSTR *pVal)
{
   return m_pFramework->get_CodeFactorySuffix(pVal);
}

STDMETHODIMP CDatabaseVP::put_CodeFactorySuffix(BSTR newVal)
{
   return m_pFramework->put_CodeFactorySuffix(newVal);
}

STDMETHODIMP CDatabaseVP::PerformTest(/*[in]*/ VARIANT Object,
     /*[optional, in]*/ VARIANT ExpectedData, /*[optional, in] */
     VARIANT ActualData, /*[out, retval]*/ enum VPResult *Result)
{
   return m_pFramework->PerformTest(Object, ExpectedData,
        ActualData, Result);
}
```

Other Responsibilities of the Verification Point Component

In addition to the implementation tasks already described in this section, your verification point component must also do the following:

- Create the `FinalConstruct()` method
- Maintain the `IsDefined` flag

These tasks are described in the following subsections:

Creating the FinalConstruct() Method

You create a `FinalConstruct()` method to initialize your verification point objects. The `FinalConstruct()` method must be defined with a `DECLARE_PROTECT_FINAL_CONSTRUCT()` statement in the class header file, as illustrated in the example in the section *The Verification Point Component* on page 173.

The `FinalConstruct()` method must perform the following tasks:

- Create the `VPFramework` object that your verification point contains.
- Put a reference to the verification point object in the `VPFramework` object.
- Initialize any properties that your verification point uses to store its metadata.
- Provide the `VPPlumbing` class with a *`ProgID`* for each component in your verification point.

The following is an example of a `FinalConstruct()` method:

```
HRESULT CDatabaseVP::FinalConstruct()
{
   HRESULT hrRetVal =
         m_pFramework.CoCreateInstance(L"RTComVP.VPFramework");
   CComQIPtr<IVPPlumbing, &IID_IVPPlumbing> plumbing;
   LPDISPATCH pTemp;
   _bstr_t bsComparator(L"RTComVP.DatabaseVPComparator");
   _bstr_t bsData(L"RTComVP.DatabaseVPData");
   _bstr_t bsDataProvider(L"RTComVP.DatabaseVPDataProvider");
   _bstr_t bsDataRenderer(L"RTComVP.DatabaseVPDataRenderer");

   m_pFramework->get_Plumbing(&pTemp);
   plumbing = pTemp;

   m_pFramework->put_VP(this);
   plumbing->InitializeFramework(bsComparator, bsData,
         bsDataProvider, bsDataRenderer);

   lOptions = 0;
   bstConnectionString = "";
   bstSQL = "";

   _com_error e(hrRetVal);
   _bstr_t bsError = e.ErrorMessage();

   return S_OK;
}
```

Maintaining the IsDefined Flag

The `VPPlumbing` class contains the boolean property *`IsDefined`*. The framework uses this property to determine if a verification point's metadata is fully specified when `PerformTest()` is invoked. If *`IsDefined`* is set to `VARIANT_FALSE`, the framework calls `DefineVP()` to prompt the tester for the missing metadata.

Your verification point implementation is responsible for coordinating the value of this property with the state of the metadata in your verification point object. The `Load()` method and the property-set methods should update the *`IsDefined`* value if they result in a change in the verification point's definition (that is, the verification point's metadata becomes fully specified or becomes no longer fully specified).

Note that the `DatabaseVP` component implements a private method named `CalcIsDefined()`. This method determines the state of the verification point's metadata and sets the *`IsDefined`* flag accordingly. All methods that might change the state of a verification point's metadata can invoke `CalcIsDefined()`.

Here is an example of *`IsDefined`* flag maintenance:

```
HRESULT CDatabaseVP::CalcIsDefined()
{
   CComQIPtr<IVPPlumbing, &IID_IVPPlumbing> plumbing;
   LPDISPATCH pTemp;

   m_pFramework->get_Plumbing(&pTemp);
   plumbing = pTemp;

   BSTR bsName;
   get_VPname(&bsName);
   _bstr_t bstName(bsName);

   BSTR bsConn;
   get_ConnectionString(&bsConn);
   _bstr_t bstConnectionString(bsConn);

   BSTR bsSQL;
   get_SQL(&bsSQL);
   _bstr_t bstSQL(bsSQL);


   if ( bstName.length() != 0 && bstConnectionString.length() != 0 &&
           bstSQL.length() != 0 )
      plumbing->put_IsDefined(VARIANT_TRUE);
   else
      plumbing->put_IsDefined(VARIANT_FALSE);

   return S_OK;
}
```

Interface for Your Verification Point Data Component

You must define an interface for your verification point data component. This interface must inherit from `IVerificationPointData`.

Your verification point data component that implements this interface contains a snapshot of the data being verified. That data can be either expected data or actual data.

The test designer should be able to use this interface to populate a Verification Point Data component for use with dynamic or manual verification points (for information, see *Types of Verification Points* on page 116).

The following is an example of an implementation of your verification point data component:

```
[
   object,
   uuid(7C4870B3-6E1A-11D4-9A26-0010A4E86989),
   dual,
   helpstring("IDatabaseVPData Interface"),
   pointer_default(unique)
]
interface IDatabaseVPData : IVerificationPointData
{
   [propget, helpstring("property NumCols")] HRESULT NumCols(
        [out, retval] long *pVal);
   [propput, helpstring("property NumCols")] HRESULT NumCols(
        [in] long newVal);
   [propget, helpstring("property NumRows")] HRESULT NumRows(
        [out, retval] long *pVal);
   [propput, helpstring("property NumRows")] HRESULT NumRows(
        [in] long newVal);
   [propget, helpstring("property Columns")] HRESULT Columns(
        [out, retval] VARIANT *pVal);
   [propput, helpstring("property Columns")] HRESULT Columns(
        [in] VARIANT newVal);
   [propget, helpstring("property Row")] HRESULT Row([in] long Index,
        [out, retval] VARIANT *pVal);
   [propput, helpstring("property Row")] HRESULT Row([in] long Index,
        [in] VARIANT newVal);
};
```

The Verification Point Data Component

Your verification point data component implements the methods defined in your verification point data interface — for example:

```
STDMETHODIMP CDatabaseVPData::get_NumCols(long *pVal)
{
   *pVal = lCols;
   return S_OK;
}

STDMETHODIMP CDatabaseVPData::put_NumCols(long newVal)
{
   lCols = newVal;
   return S_OK;
}

STDMETHODIMP CDatabaseVPData::get_NumRows(long *pVal)
{
   *pVal = lRows;
   return S_OK;
}

STDMETHODIMP CDatabaseVPData::put_NumRows(long newVal)
{
   lRows = newVal;
   return S_OK;
}

STDMETHODIMP CDatabaseVPData::get_Columns(VARIANT *pVal)
{

   VariantInit(pVal);
   pVal->vt = (VT_ARRAY | VT_BYREF | VT_BSTR);

   ...
   // Copy the array and return it to the caller.
   ...
   return S_OK;
}

STDMETHODIMP CDatabaseVPData::put_Columns(VARIANT newVal)
{
   pvColumns = new VARIANT;
   VariantInit(pvColumns);
   pvColumns->vt = (VT_ARRAY | VT_BSTR);
   SafeArrayCopy(newVal.parray, &(pvColumns->parray));
   put_NumCols(newVal.parray->rgsabound[0].cElements);

   return S_OK;
}

STDMETHODIMP CDatabaseVPData::get_Row(long Index, VARIANT *pVal)
{
```

```
   VariantInit(pVal);
   pVal->vt = (VT_ARRAY | VT_BYREF | VT_BSTR);

   if ( paRows != NULL )
   {
      ...
      // Copy the row into the output array
      ...
   }
   return S_OK;
}

STDMETHODIMP CDatabaseVPData::put_Row(long Index, VARIANT newVal)
{
   ...
   // Copy this row into our data structure
   ...

   return S_OK;
}
```

In addition, your verification point data component must implement the following:

- The `IPersistFile` interface to provide for its own serialization
- The `FileExtension()` property methods

The following sections describe these tasks.

Implementing the IPersistFile Interface

Your verification point class must implement its own serialization to a verification point data file by implementing the `IPersistFile` interface.

As with your verification point class, you only need to implement the `Load()` and `Save()` methods, and you may return `E_NOTIMPL` for the other methods.

Because the Grid Comparator in the current version of the product also accesses your metadata and data files, you must use a .ini metafile format and a .csv data file format. In a future release, you will be able to create your own comparator applications to read from and write to files of any data file format you choose.

The following is an example of data file serialization:

```
STDMETHODIMP CDatabaseVPData::GetClassID(CLSID *pClassID)
{
   return E_NOTIMPL;
}

STDMETHODIMP CDatabaseVPData::IsDirty()
{
   return E_NOTIMPL ;
}
```

```
STDMETHODIMP CDatabaseVPData::SaveCompleted(LPCOLESTR pszFileName)
{
   return E_NOTIMPL ;
}

STDMETHODIMP CDatabaseVPData::GetCurFile(LPOLESTR *ppszFileName)
{
   return E_NOTIMPL;
}

STDMETHODIMP CDatabaseVPData::Load(LPCOLESTR pszFileName,
        DWORD dwMode)
{
   SAFEARRAY *psaColumns = NULL;
   SAFEARRAY *psaRow = NULL;
   wstring buffer;
   HRESULT hr = S_OK;
   wchar_t delim = L'\n';
   long lNumVals = 0;

   _bstr_t bsFile = pszFileName;
   wifstream stream(bsFile);

   getline(stream, buffer, delim);
   if ( buffer.empty() != true )
   {
      lNumVals = GetNumElementsFromCSVString( buffer.c_str() );

      hr = BuildBSTRSafeArrayFromCSVString( buffer.c_str(),
           &psaColumns, lNumVals, 0 );
      if ( hr == S_OK )
      {
         pvColumns = new VARIANT;
         VariantInit(pvColumns);

         pvColumns->vt = (VT_ARRAY | VT_BSTR);
         pvColumns->parray = psaColumns;
         put_NumCols(lNumVals);
      }
      else
      {
         return E_INVALIDARG;
      }
   }
   else
   {
      put_NumCols(0);
      put_NumRows(0);
      return S_OK;
   }

   getline(stream, buffer, delim);
   for ( long l = 0; true != buffer.empty(); l++ )
```

```
      {
         hr = BuildBSTRSafeArrayFromCSVString( buffer.c_str(), &psaRow,
               lNumVals, 0 );
         if ( hr == S_OK )
         {
            VARIANT vRow;
            VariantInit(&vRow);
            vRow.vt = (VT_ARRAY | VT_BSTR);
            vRow.parray = psaRow;
            put_Row(l, vRow);
            VariantClear( &vRow );
         }
         else
         {
            return E_INVALIDARG;
         }
         getline(stream, buffer, delim);
      }

      return S_OK;
   }

   STDMETHODIMP CDatabaseVPData::Save(LPCOLESTR pszFileName,
          BOOL fRemember)
   {
      SAFEARRAY *psaColumns = NULL;
      SAFEARRAY *psaRow = NULL;
      FILE *pfOut = NULL;

      // Validate parameters
      if ( pszFileName == NULL )
         return E_POINTER;

      if ( *pszFileName == L'\0' )
         return E_INVALIDARG;

      if ( pvColumns == NULL || pvColumns->vt != (VT_ARRAY | VT_BSTR) )
         return E_INVALIDARG;

      psaColumns = pvColumns->parray;

       // If there's nothing to write -- don't write anything...
      lCols = psaColumns->rgsabound[0].cElements;

       if ( lCols == 0 )
           return S_OK;

      // Open the file
      pfOut = _wfopen(pszFileName, L"wt");

      if  ( pfOut == NULL )
         return E_INVALIDARG;

      // Write out the file!
```

```
   // First print out a line with all the column names.
   WriteBSTRSafeArrayToCSVFile(pfOut, psaColumns);

    for ( long l=0; l < lRows; l++ )
    {
     psaRow = paRows[l].parray;

     if ( psaRow != NULL)
        WriteBSTRSafeArrayToCSVFile(pfOut, psaRow);

    }

   fclose(pfOut);
   return S_OK;
}
```

Implementing the FileExtension() Property Methods

Your verification point data component must implement the following `FileExtension()` property methods:

- `get_FileExtension`
- `put_FileExtension`).

In the current release of Rational QualityArchitect, set this property to `csv`. In a future release, this property should contain the file extension associated with the data file format used by your verification point data component — for example, `csv`, `dat`, or `xml`.

The verification point framework creates a unique data file name and passes it to the `Load()` and `Save()` methods. The `FileExtension()` property method tells the framework the file extension to use for this verification point data type.

The Verification Point Data Comparator Component

Your specialized Verification Point Data Comparator component must implement the `IVerificationPointComparator` interface. This interface has only one method, `Compare()`.

The `Compare()` method compares an expected data object and an actual data object, both of type I`VerificationPointData`, and determines whether the test succeeds or fails.

The following is an example of a data comparison:

```
STDMETHODIMP CDatabaseVPComparator::Compare(
      /*[in]*/ IVerificationPointData *ExpectedData,
      /*[in]*/ IVerificationPointData *ActualData,
      /*[in]*/ VARIANT Options, /*[out]*/ BSTR *FailureDescription,
      /*[out, retval]*/  VARIANT_BOOL *Result)
{
   CComQIPtr<IDatabaseVPData, &IID_IDatabaseVPData> vpdExpected;
   CComQIPtr<IDatabaseVPData, &IID_IDatabaseVPData> vpdActual;

   vpdExpected = ExpectedData;
   vpdActual = ActualData;
   *Result = VARIANT_FALSE;
   long lNumCols = 0;
   bool bCaseInsensative = false;
   BSTR bsFoo;

   // Allow for NULL FailureDescription
   if ( FailureDescription == NULL )
   {
      // assign to throwaway local
      FailureDescription = &bsFoo;
   }

   // First compare the column names.
   VARIANT vExpColumns;
   VARIANT vActColumns;
   SAFEARRAY *psaExpColumns;
   SAFEARRAY *psaActColumns;

   vpdExpected->get_Columns(&vExpColumns);
   vpdActual->get_Columns(&vActColumns);

   if ( vExpColumns.vt == (VT_ARRAY | VT_BYREF | VT_BSTR) &&
            vActColumns.vt == (VT_ARRAY | VT_BYREF | VT_BSTR))
   {
      psaExpColumns = *(vExpColumns.pparray);
      psaActColumns = *(vActColumns.pparray);

      lNumCols = psaExpColumns->rgsabound[0].cElements;
      if ((unsigned long)lNumCols !=
                psaActColumns->rgsabound[0].cElements)
      {
         *FailureDescription = SysAllocString(L"Expected and Actual
              resultsets had different number of columns.");
         return S_OK;
      }


      if ( !CompareBstrSafeArray(psaExpColumns, psaActColumns,
           NULL, bCaseInsensative))
      {
         *FailureDescription = SysAllocString(L"Expected and Actual
```

```
            resultsets had different column names.");
      return S_OK;
   }
}


// Now loop over each of the adjacent rows and compare them.

long lNumExpRows = 0;
long lNumActRows = 0;
vpdExpected->get_NumRows(&lNumExpRows);
vpdActual->get_NumRows(&lNumActRows);

if ( lNumExpRows != lNumActRows )
{
   *FailureDescription = SysAllocString(L"Expected and Actual
        resultsets had different number of rows.");
   return S_OK;
}


VARIANT vExpRow;
VARIANT vActRow;
SAFEARRAY *psaExpRow;
SAFEARRAY *psaActRow;

for ( long lIndex = 0; lIndex < lNumExpRows; lIndex++ )
{

   vpdExpected->get_Row(lIndex, &vExpRow);
   vpdActual->get_Row(lIndex, &vActRow);


   if ( vExpRow.vt == (VT_ARRAY | VT_BYREF | VT_BSTR) &&
       vActRow.vt == (VT_ARRAY | VT_BYREF | VT_BSTR))
   {
      psaExpRow = *(vExpRow.pparray);
      psaActRow = *(vActRow.pparray);

      if ( !CompareBstrSafeArray(psaExpRow, psaActRow,
           FailureDescription, bCaseInsensative))
      {
         // FailureDescription filled in by CompareBstr rountine.
         return S_OK;
      }

   }
   else
   {
      ...
      // Problem with data objects -- handle error
      ...
   }
```

```
   }

   *Result = VARIANT_TRUE;
   return S_OK;
}
```

The Verification Point Data Provider Component

Your specialized Verification Point Data Provider component must implement the `IVerificationPointDataProvider` interface. This interface has only one method, `CaptureData()`.

The `CaptureData()` method reads the verification point's definition from the supplied verification point object, captures the data required by the verification point, and returns the data in a new `IVerificationPointData` object.

The following example illustrates an implementation of the `IVerificationPointDataProvider` interface:

```
STDMETHODIMP CDatabaseVPDataProvider::CaptureData (
     /*[in]*/ VARIANT Object, /*[in]*/ IVerificationPoint *VP,
     /*[out, retval]*/ IVerificationPointData **Data)
{

   //  QI for DatabaseVP interface
   ...

   dbVP->get_ConnectionString(&bsConnection);
   _bstr_t bstConnection(bsConnection, false);
   dbVP->get_SQL(&bsSQL);
   _bstr_t bstSQL(bsSQL, false);


   // Attempt to connect to the OLE DB using the connection string
   // stored in the VP object.
   ...

   long lNumCols = rs->Fields->Count;

   if ( lNumCols > 0 )
   {
      CComQIPtr<IDatabaseVPData, &IID_IDatabaseVPData> dataReturn;
      _bstr_t bstDataProgID = "rtComVP.DatabaseVPData";
      dataReturn.CoCreateInstance(bstDataProgID);


      VARIANT v;
      v.vt = VT_I4;

      SAFEARRAYBOUND rgsaBound[1];
      rgsaBound[0].lLbound = 0;
      rgsaBound[0].cElements = lNumCols;
      SAFEARRAY *psaColumns = SafeArrayCreate( VT_BSTR, 1, rgsaBound );
```

```
      for ( long l=0; l < lNumCols; l++ )
      {
         v.lVal = l;
         BSTR bsColumn = SysAllocString(rs->Fields->Item[v]->Name);
         SafeArrayPutElement( psaColumns, &l, bsColumn);
      }

      VARIANT vColumns;
      VariantInit(&vColumns);
      vColumns.vt = (VT_ARRAY | VT_BSTR);
      vColumns.parray = psaColumns;

      dataReturn->put_Columns(vColumns);

      rs->MoveLast();
      long lNumRows = rs->RecordCount;
      rs->MoveFirst();

      dataReturn->put_NumRows(lNumRows);

      for ( l=0; !rs->EOF; l++,rs->MoveNext() )
      {
         SAFEARRAY *psaRow = SafeArrayCreate( VT_BSTR, 1, rgsaBound );
         for ( long j = 0; j < lNumCols; j++ )
         {
            v.lVal = j;
            _bstr_t Temp = rs->Fields->Item[v]->Value;
            BSTR bsData = SysAllocString(Temp);
            SafeArrayPutElement( psaRow, &j, bsData );
         }

         VARIANT vRow;
         VariantInit(&vRow);
         vRow.vt = (VT_ARRAY | VT_BSTR);
         vRow.parray = psaRow;
         dataReturn->put_Row(l, vRow);
      }

      dataReturn->QueryInterface(IID_IVerificationPointData,
           (void **) Data);
      (*Data)->AddRef();

   }

   // Clean up memory
   ...

   return S_OK;

}
```

The Verification Point Data Renderer Component

Your specialized Verification Point Data Renderer component must implement the `IVerificationPointDataRenderer` interface. This interface has only one method, `DisplayAndValidateData()`.

`DisplayAndValidateData()` displays the data in an `IVerificationPointData` object, allowing the tester to accept or reject that data as being correct.

The framework calls this method when both of the following conditions exist:

- A static verification point is invoked for the first time (that is, when the expected data is first captured).
- The test designer has set `VPOPTION_USER_ACKNOWLEDGE_BASELINE` in the *`Options`* property of the `IVerificationPoint` component.

If the tester accepts the displayed data, the data is stored as the expected data for the static verification point. If the tester rejects the data, no expected data is stored, and the process is repeated the next time the verification point is executed.

The following example illustrates an implementation of the `IVerificationPointDataRenderer` interface:

```
STDMETHODIMP CDatabaseVPDataRenderer::DisplayAndValidateData(
      /*[in, out]*/ IVerificationPointData **Data,
      /*[out, retval]*/ VARIANT_BOOL *Valid)
{

   CComQIPtr<_clsDatabaseVPDataRenderer,
         &IID__clsDatabaseVPDataRenderer> pRend;
   CComQIPtr<IDatabaseVPData, &IID_IDatabaseVPData> pDBdata;
   VARIANT vCols;
   VARIANT vRow;
   VARIANT vAccepted;

   // Create an instance of the GUI data renderer.
   pRend.CoCreateInstance(L"rtCOMVpGui.clsDatabaseVPDataRenderer");

   // Get a DatabaseVPData COM pointer.
   pDBdata = *Data;

   if ( pDBdata == NULL || pRend == NULL)
   {
      return S_FALSE;
   }

   // Put the columns from the data object into the GUI.
   pDBdata->get_Columns(&vCols);
   pRend->put_Columns(vCols);

   long lNumRows;
```

```
   pDBdata->get_NumRows(&lNumRows);

   // Put the rows from the data object into the GUI.
   for ( long l = 0; l < lNumRows; l++ )
   {
      pDBdata->get_Row( l, &vRow );
      pRend->put_Row( vRow );
   }

   // Invoke the GUI dialog.
   pRend->DoModal();

   // Pass back the result.
   pRend->get_Accepted(&vAccepted);

   if ( vAccepted.vt == VT_BOOL && vAccepted.boolVal == VARIANT_TRUE )
   {
      *Valid = VARIANT_TRUE;
   }
   else
   {
      *Valid = VARIANT_FALSE;
   }

   return S_OK;
}
```

Integrating Your Verification Point with QualityArchitect

Once you have implemented a verification point, you should 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.

To integrate your verification point with QualityArchitect, do the following:

1. Register the verification point in the rqalocvp.ini file. This file lists custom verification point types in the section COM VP in the format *vptype = progID* — for example:

   ```
   [COM VP]
   DatabaseVP=RTComVP.DatabaseVP
   ```

   The rqalocvp.ini file is located in the Rational datastore in the folder DefaultTestScriptDataStore.

2. Register the .dll file containing your verification point component.

IDL Equivalents

D

This appendix presents the verification point methods in Information Definition Language (IDL) format. You might find IDL format useful if you are implementing new verification point types in C++.

```
import "oaidl.idl";
import "ocidl.idl";
import "..\..\CTDatastore\CTDatastore.idl";

import "..\..\..\src\shlib\dcom\sqavuservices\vuservices.idl";

   [
      object,
      uuid(F1DCD5A5-4F40-11D4-99DE-000000000000),
      dual,
      helpstring("IVerificationPointData Interface"),
      pointer_default(unique)
   ]

   interface IVerificationPointData : IDispatch
   {

   [propget, id(1), helpstring("This property specifies the file
   extension used by the VerificationPointData's disk representation.
   The correct extension is necessary for correct Comparator
   behavior.")]
   HRESULT FileExtension([out, retval] BSTR *pVal);

   [propput, id(1), helpstring("This property specifies the file
   extension used by the VerificationPointData's disk representation.
   The correct extension is necessary for correct Comparator
   behavior.")]
   HRESULT FileExtension([in] BSTR newVal);
   };

   [
      object,
      uuid(F1DCD5AA-4F40-11D4-99DE-000000000000),
      dual,
      helpstring("IVerificationPointComparator Interface"),
      pointer_default(unique)
   ]

   interface IVerificationPointComparator : IDispatch
   {
```

```
[id(1), helpstring("This method compares two objects impementing the
IVerificationPointData interface.  It should be invoked only by the
VP framework.")]
HRESULT Compare([in] IVerificationPointData *ExpectedData,
   [in] IVerificationPointData *ActualData, [in] VARIANT Options,
   [in, out] BSTR *FailureDescription,
   [out, retval]  VARIANT_BOOL *Result);
};

[
   object,
   uuid(F1DCD5AC-4F40-11D4-99DE-000000000000),
   dual,
   helpstring("IVerificationPointDataRenderer Interface"),
   pointer_default(unique)
]

interface IVerificationPointDataRenderer : IDispatch
{

[id(1), helpstring("method DisplayAndValidateData")]
HRESULT DisplayAndValidateData(
   [in, out] IVerificationPointData **Data,
   [out, retval] VARIANT_BOOL *Valid);
};

[
   object,
   uuid(3E21F5BA-B4FF-46C2-9E35-8A784497DC91),
   dual,
   helpstring("IVPFramework Interface"),
   pointer_default(unique)
]

interface IVPFramework : IDispatch
{

[propget, id(1), helpstring("The name of the Verification Point")]
HRESULT VPname([out, retval] BSTR *pVal);

[propput, id(1), helpstring("The name of the Verification Point")]
HRESULT VPname([in] BSTR newVal);

[id(2), helpstring("This method performs the verification.  The
default verification is static.  To perform a dynamic verification,
pass an expected data object.  To perform a manual verification,
pass expected and actual data objects.")]
HRESULT PerformTest([in] VARIANT Object,
   [in,optional] VARIANT ExpectedData, [in,optional] VARIANT
   ActualData, [out, retval] enum VPResult *Result );

[hidden, propget, id(3), helpstring("For internal use only.")]
HRESULT VP ( [out, retval] LPDISPATCH *pVP );
```

```
[hidden, propput, id(3), helpstring("For internal use only.")]
HRESULT VP ( [in] LPDISPATCH newVP );

[hidden, propget, id(4), helpstring("For internal use only.")]
HRESULT Plumbing([out, retval] LPDISPATCH *pVal);

[hidden, propput, id(4), helpstring("For internal use only.")]
HRESULT Plumbing([in] LPDISPATCH newVal);

[hidden, propget, id(5), helpstring("A unique identifer to append to
a VP's code factory variable names.  This allows the code factory
methods to prevent name collisions when multiple VPs are created in
the same scope.")]
HRESULT CodeFactorySuffix([out, retval] BSTR *pVal);

[hidden, propput, id(5), helpstring("A unique identifer to append to
a VP's code factory variable names.  This allows the code factory
methods to prevent name collisions when multiple VPs are created in
the same scope.")]
HRESULT CodeFactorySuffix([in] BSTR newVal);
};

[
   object,
   uuid(F1DCD5A3-4F40-11D4-99DE-000000000000),
   dual,
   helpstring("IVerificationPoint Interface"),
   pointer_default(unique)
]

interface IVerificationPoint : IVPFramework
{

[hidden, id(15), helpstring("This method invokes a GUI to capture
the VP's definition.")]
HRESULT DefineVP();

[hidden, id(16), helpstring("This method returns a syntactically
valid constructor invocation (for a given language) which can be
inserted into a recorded or generated script.")]
HRESULT CodeFactoryGetConstructorInvocation([in] CTDScriptTypes
   Language, [out, retval] BSTR *Code);

[hidden, id(17), helpstring("This method returns the number of
externalized input variables required for code generation of this
VP for a given language.")]
HRESULT CodeFactoryGetNumExternallizedInputs([in] CTDScriptTypes
   Language, [out, retval] short *NumInputs);

[hidden, id(18), helpstring("This method returns a line of
syntactically correct code (for a given language) declaring the nth
externalized input for this VP.")]
HRESULT CodeFactoryGetExternalizedInputDecl([in] CTDScriptTypes
```

```
   Language, [in] short InputNumber, [out, retval] BSTR *Code);

[hidden, id(19), helpstring("This method returns a line of
syntactically correct code (for a given language) initializing the
nth externalized input for this VP.")]
HRESULT CodeFactoryGetExternalizedInputInit([in] CTDScriptTypes
   Language, [in] short InputNumber, [out, retval] BSTR *Code);

[hidden, id(20), helpstring("This method returns the number of
property set calls required to fully specify this VP's definition
for code generation of this VP for a given language.")]
HRESULT CodeFactoryGetNumPropertySet([in] CTDScriptTypes Language,
   [out, retval] short *NumProps);

[hidden, id(21), helpstring("This method returns a line of
syntactically correct code (for a given language) setting the nth
property for this VP.")]
HRESULT CodeFactoryGetPropertySet([in] CTDScriptTypes Language,
   [in] short InputNumber, [out, retval] BSTR *Code);

[propget, id(22), helpstring("This property stores any Options which
affect the behavior of the DataProvider or the Comparator.")]
HRESULT Options([out, retval] VARIANT *pVal);

[propput, id(22), helpstring("This property stores any Options which
affect the behavior of the DataProvider or the Comparator.")]
HRESULT Options([in] VARIANT newVal);
};

[
   object,
   uuid(F1DCD5A8-4F40-11D4-99DE-000000000000),
   dual,
   helpstring("IVerificationPointDataProvider Interface"),
   pointer_default(unique)
]

interface IVerificationPointDataProvider : IDispatch
{

[id(1), helpstring("This method reads the VP's definition from the
supplied VP object, captures the data required by the VP, and
returns that data in a new IVerificationPointData object.")]
HRESULT CaptureData([in] VARIANT Object,[in] IVerificationPoint *VP,
   [out, retval] IVerificationPointData **Data);
};

[
   object,
   uuid(15937740-5F7E-11d4-9A07-000000000000),
   dual,
   helpstring("IVPPlumbing Interface"),
   pointer_default(unique)
]
```

```
interface IVPPlumbing : IDispatch
{

[/*id(6),*/ helpstring("This method informs the VP Framework of the
helper components used by this VP type.")]
HRESULT InitializeFramework([in] BSTR VPComparator,
   [in] BSTR VPData, [in] BSTR VPDataProvider,
   [in] BSTR VPDataRenderer);

[/*id(1),*/ helpstring("This method deserializes the VP from the
repo if necessary, calls the VPs defineVP method if required, and
serializes the resulting VP definition.")]
HRESULT InitializeVP();

[propget, id(1), helpstring("This property specifies whether or not
the VP has been fully specified.  An incompletely specified VP will
have defineVP invoked by the Framework.")]
HRESULT IsDefined([out, retval] VARIANT_BOOL *pVal);

[propput, id(1), helpstring("This property specifies whether or not
the VP has been fully specified.  An incompletely specified VP will
have defineVP invoked by the Framework.")]
HRESULT IsDefined([in] VARIANT_BOOL newVal);

[propget, id(2), helpstring("This property specifies whether or not
the VP is in a valid state for PerformTest to be invoked.")]
HRESULT IsValid([out, retval] VARIANT_BOOL *pVal);

[propput, id(2), helpstring("This property specifies whether or not
the VP is in a valid state for PerformTest to be invoked.")]
HRESULT IsValid([in] VARIANT_BOOL newVal);

[propget, id(3), helpstring("This property contains the ProgID of
the VPComparator class for this VP")]
  HRESULT VPComparator([out, retval] BSTR *pVal);

[propput, id(3), helpstring("This property contains the ProgID of
the VPComparator class for this VP")]
HRESULT VPComparator([in] BSTR newVal);

[propget, id(4), helpstring("This property contains the ProgID of
the VPData component for this VP")]
HRESULT VPData([out, retval] BSTR *pVal);

[propput, id(4), helpstring("This property contains the ProgID of
the VPData component for this VP")]
HRESULT VPData([in] BSTR newVal);

[propget, id(5), helpstring("This property contains the
VPDataProvider component for this VP.")]
HRESULT VPDataProvider([out, retval] BSTR *pVal);

[propput, id(5), helpstring("This property contains the
```

```
VPDataProvider component for this VP.")]
   HRESULT VPDataProvider([in] BSTR newVal);

   [propget, id(6), helpstring("This property contains the
   VPDataRenderer component for this VP.")]
   HRESULT VPDataRenderer([out, retval] BSTR *pVal);

   [propput, id(6), helpstring("This property contains the
   VPDataRenderer component for this VP.")]
   HRESULT VPDataRenderer([in] BSTR newVal);
   };

   [
      object,
      uuid(7C4870B0-6E1A-11D4-9A26-0010A4E86989),
      dual,
      helpstring("IDatabaseVP Interface"),
      pointer_default(unique)
   ]

   interface IDatabaseVP : IVerificationPoint
   {

   [propget, helpstring("property ConnectionString")]
   HRESULT ConnectionString([out, retval] BSTR *pVal);

   [propput, helpstring("property ConnectionString")]
   HRESULT ConnectionString([in] BSTR newVal);

   [propget, helpstring("property SQL")]
   HRESULT SQL([out, retval] BSTR *pVal);

   [propput, helpstring("property SQL")]
   HRESULT SQL([in] BSTR newVal);
   };

   [
      object,
      uuid(7C4870B3-6E1A-11D4-9A26-0010A4E86989),
      dual,
      helpstring("IDatabaseVPData Interface"),
      pointer_default(unique)
   ]

   interface IDatabaseVPData : IVerificationPointData
   {

   [propget, helpstring("property NumCols")]
   HRESULT NumCols([out, retval] long *pVal);

   [propput, helpstring("property NumCols")]
   HRESULT NumCols([in] long newVal);

   [propget, helpstring("property NumRows")]
```

```
   HRESULT NumRows([out, retval] long *pVal);

   [propput, helpstring("property NumRows")]
   HRESULT NumRows([in] long newVal);

   [propget, helpstring("property Columns")]
   HRESULT Columns([out, retval] VARIANT *pVal);

   [propput, helpstring("property Columns")]
   HRESULT Columns([in] VARIANT newVal);

   [propget, helpstring("property Row")]
   HRESULT Row([in] long Index, [out, retval] VARIANT *pVal);

   [propput, helpstring("property Row")]
   HRESULT Row([in] long Index, [in] VARIANT newVal);
   };

[
   uuid(20346813-4073-11D4-99CD-0010A4E86989),
   version(1.0),
   helpstring("Rational QualityArchitect COM Verification Point
      Interface Type Library")
]

library RTIVP
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   enum VPResult
   {
      VERIFICATION_NO_RESULT = 0, // TSS_LOG_RESULT_NONE
      VERIFICATION_SUCCEEDED = 1, // TSS_LOG_RESULT_PASS
      VERIFICATION_FAILED    = 2, // TSS_LOG_RESULT_FAIL
      VERIFICATION_ERROR     = 3, // TSS_LOG_RESULT_WARN
   };

   enum VPOptions
   {
   /** Specifies that the verification should be case insensitive. */
      VPOPTION_COMPARE_CASEINSENSITIVE = 1,

   /** 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. */
      VPOPTION_USER_ACKNOWLEDGE_BASELINE = 2,
      VPOPTION_EXPECT_FAILURE = 4
   };

   enum DatabaseVPOptions
   {
      DATABASEVPOPTION_TRIM_WHITESPACE = 8
   };
```

```
    interface IVerificationPoint;
    interface IVerificationPointData;
    interface IVerificationPointDataProvider;
    interface IVerificationPointDataRenderer;
    interface IVerificationPointComparator;
    interface IVPFramework;
    interface IVPPlumbing;
    interface IDatabaseVP;
    interface IDatabaseVPData;
};
```

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W