JavaTest Harness

Architect's Guide

Part No.: JTAG
May 2011

Please
Recycle

Contents

Figures

Tables

Preface

This manual is intended for test suite architects who design JavaTest harness test suites. It assumes that you are familiar with the Java programming language and with running Java programs on at least one platform implementation.

Before You Read This Book

It is highly recommended that you read the JavaTest online help, the *Test Suite Developer's Guide*, and *TCK Project Planning and Development Guide*, which are available as part of the Java Compatibility Test Tools release. Note that for convenience, the JavaTest online help is also available in PDF format

This guide is divided into the following chapters and appendices:

Chapter 1	Introduction	
Part I	The Basics	
	Chapter 2	A tutorial that introduces the JavaTest GUI.
	Chapter 3	Describes the test suite components for which architects are responsible.
	Chapter 4	Leads you through the process of creating a small working test suite.
	Chapter 5	Describes how to write tests that work well with JavaTest Harness.
	Chapter 6	Describes how to create configuration interviews for test suites.

Part II	Advanced Topics	
	Chapter 7	Describes how to use the JavaTest Harness to compile test suites.
	Chapter 8	Describes how test finders work and how to create a customized version for your test suite.
	Chapter 9	Describes how test scripts work and how to create a customized version for your test suite.
	Chapter 10	Describes how the test suite object works and how to create a customized version for your test suite.
	Chapter 11	Describes the ServiceManager component provided by JavaTest Harness and how test suite architects can use it to manage services.
	Chapter 12	Describes how to retrofit existing JUnit 3.x or 4.x test suites to enable them to run with the JavaTest Harness harness.
	Chapter 13	Describes customizations that test suite architects can make in the JavaTest Harness harness.
	Appendix A	Describes the standard commands available from the JavaTest command library.
	Appendix B	Describes the file formats and specifications used by JavaTest Harness.
	Appendix C	Tips for writing interviews.
Glossary		Defines terms used in this book and other TCK documentation.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files.
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
Bourne shell and Korn shell	`%`
MSDOS	C:\>

Related Documentation

Technology	Title
JavaTest harness	*JavaTest online help* (available both online and in PDF format)
TCK development process	TCK Project Planning and Development Guide
Java Compatibility Test Tools	*Test Suite Developer's Guide*

Accessing Java Platform Documentation Online

The Oracle Technology Network enables you to access Java SE or Java ME technical documentation on the Web:

http://download.oracle.com/javase/index.html

http://download.oracle.com/javame/index.html

Oracle Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

javasedocs_us@oracle.com or javamedocs_us@oracle.com

CHAPTER 1

Introduction

A Technology Compatibility Kit (TCK) is a test suite and a set of tools used to certify that an implementation of a Java technology conforms both to the applicable Java platform specifications and to the corresponding reference implementations — the end result is a Java technology that is certified as compatible. The architect designs test suites to exercise assertions described in the technology specifications. TCK test suites may use the JavaTest harness for test execution and test suite management tools.

As the test suite architect, it is your job to design the framework for the test suite, and if necessary, create custom plug-in components required to implement the design.

Each TCK that you design should be composed of the following:

Test suite: A collection of tests that fit within a common framework. The framework is typically designed by the architect — the individual tests are designed to work within the framework and are usually written by a team of test suite developers.

JavaTest harness: The test harness used to run the tests in the test suite. You (the architect) may have to provide plug-in components that know how to identify and execute the tests.

Configuration interview: To run the tests in a test suite, the JavaTest harness requires site-specific information about the computing environment, such as the location of the Java launcher, and the Java technology being tested. The JavaTest harness provides the means to collect this information based on an interview that you provide.

Documentation: A well documented TCK generally includes the following information, provided by the architect:

- **Test Suite User's Guide**: Contains instructions about how to start and run the tests and rules for certification. The Java Compatibility Test Tools (JCTT) release contains a *TCK User's Guide Template* that can serve as the basis for this document.

- **Configuration editor "More Info" text**: Provides explanation and examples for each question in the configuration interview

Architects design test suites and the characteristics of the various tests, but are not typically concerned with the specific details of individual tests. That is the task of test suite developers (see the *Test Suite Developer's Guide*). Architects design the framework in which the individual tests fit.

This document describes the tasks associated with the TCK architect.

Examples

The `examples` directory contains example test suites that are used throughout this book in tutorials and to illustrate how tests and test suites are constructed. Please use these examples to supplement the discussions in this manual. The `examples` directory contains the following:

```
...\examples\
     javatest\
        demoapi.jar      API classes tested by the Demo TCK test suite
        interviewDemo\   A self-documenting configuration interview demo
             demotck\    The test suite used to run the interview demo
             src         The interview demo source files
        simpleHTML\
             demotck\    Demo test suite that uses HTML-based test
             src         Demo TCK configuration interview source files
        simpleTags\
             demotck\    Demo test suite that uses HTML-based test
             src         Demo TCK configuration interview source files
        sampleFiles\     Miscellaneous sample source files in this manual.
```

Note – Unless otherwise indicated, all examples in this book use Microsoft Windows style command prompts and file separators.

PART I The Basics

The chapters in this part of the JavaTest Architect's Guide introduce the JavaTest GUI, basic concepts, and provide enough information to create a basic test suite.

CHAPTER 2

JavaTest Tutorial

This tutorial introduces you to the JavaTest version GUI and some basic underlying concepts. The tutorial instructions have you run a very simple test suite called Demo TCK that is included in the `examples` directory. Demo TCK contains 17 tests that test the functionality of some very simple demo APIs.

This tutorial touches only on the core functionality of the JavaTest harness GUI. Please consult the online help for information about all of the JavaTest features.

Overview

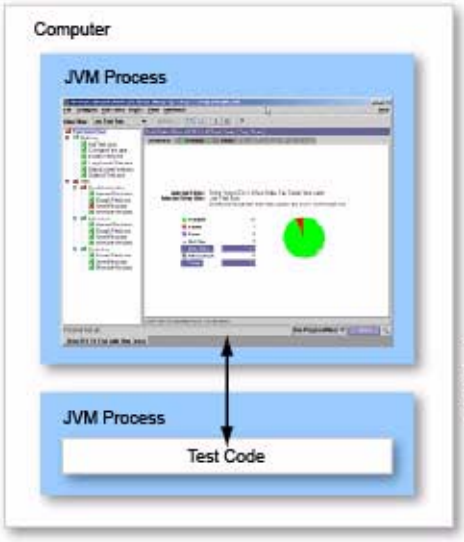
The tutorial should be run using version 6.0 or later of the Java Platform, Standard Edition (Java SE) on either the Solaris Operating System (Solaris OS) or the Microsoft Windows (WIN32) operating system.

Note – Unless otherwise indicated, all examples in this book use Microsoft Windows style command prompts and file separators.

To keep things simple, these instructions show you how to run both the JavaTest harness and the tests on the same system in different[1] Java virtual machines (JVM processes[2]). FIGURE 2-1 diagram illustrates this point.

1. It is also possible to run the JavaTest harness and the tests on separate systems using the JavaTest Agent.
2. The terms “Java virtual machine” and “JVM” are sometimes used to mean “a virtual machine for the Java platform”.

FIGURE 2-1 JavaTest Harness and Tests Running on Same System



Running the Tutorial

The tutorial tasks are as follows:

1. Start the JavaTest Harness
2. Run the Quick Start Wizard
3. Configure Test Information
4. Run Tests
5. Browse the Results
6. Exclude the Failed Test
7. Generate a Report

▼ Start the JavaTest Harness

1. **Verify that the Java SE platform (version 1.5 or later) is in your path.**

   At a command prompt, enter:

   `C:\>` **`java -version`**

2. **Make** *jt_install*`\examples\javatest\simpleTags\demotck` **the current directory.**

   The directory *jt_install* is the directory into which you installed the JavaTest harness software.

3. **Start the JavaTest harness.**

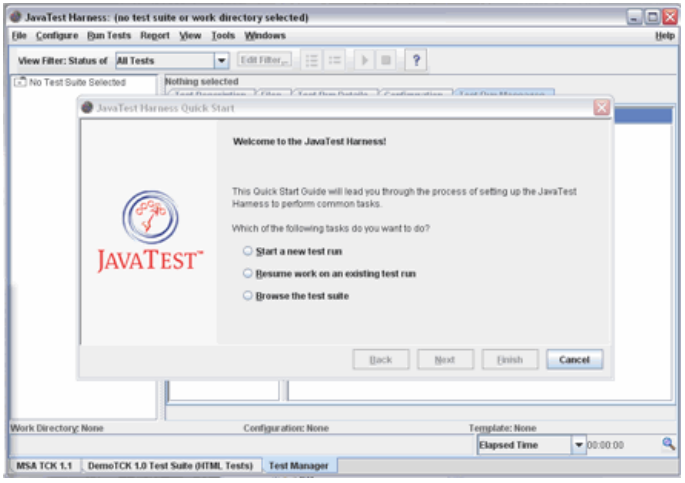
   At a command prompt enter:

   `C:\>` **`java -jar lib\javatest.jar -newDesktop`**

Note – The `-newDesktop` option is used here to ensure that the JavaTest harness starts up exactly as described in these instructions — under normal circumstances you should *not* use this option because you will lose information that the harness saved about your previous session. For information about JavaTest options, see the JavaTest online help.

The JavaTest harness should start and display the Quick Start wizard window:

FIGURE 2-2 The JavaTest Harness with Quick Start Wizard



▼ Run the Quick Start Wizard

The Quick Start wizard leads you through the basic steps required to start running the test suite.

1. **Panel 1: Welcome to the JavaTest Harness**

   Choose "Start a new test run", and click Next

2. **Panel 2: Test Suite**

   Click the Next button (accept the default).

3. **Panel 3: Configuration**

   Choose "Create a new configuration", and click Next

4. **Panel 4: Work Directory**

   The JavaTest harness uses the work directory to store information and to write test results. Click the Browse button to activate a file chooser. Use the file chooser to create a work directory — be sure to create the work directory in a convenient location *outside of the test suite directory* (`demotck`). Click Next.

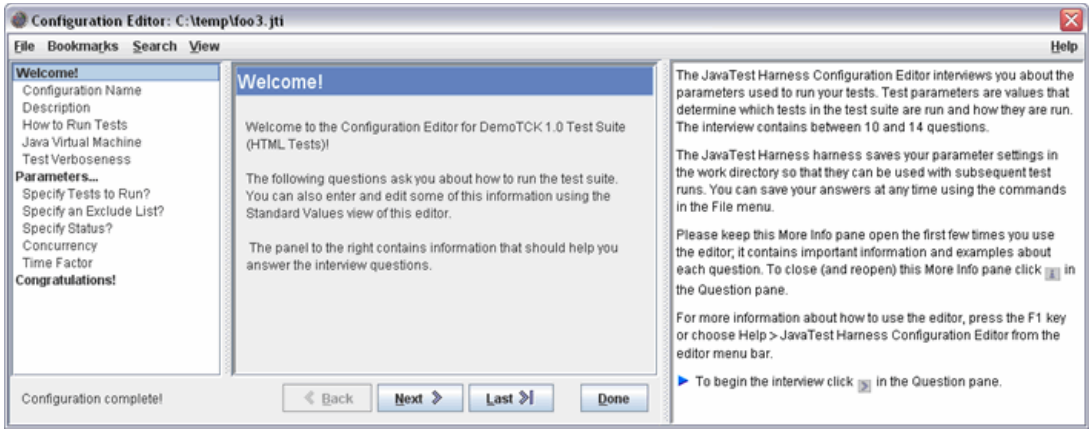
5. **Panel 5: Almost Done ...**

   Click the Finish button to complete the Quick Start process. For these options, the configuration editor window is started automatically.

▼ Configure Test Information

Because the "Start the configuration editor" checkbox was checked in the last panel of the Quick Start wizard, the configuration editor starts automatically.

You use the configuration editor to configure the information required to run the test suite. As shown below, the configuration editor consists of three panes and a menu bar:



The left pane lists the titles of the questions you have answered, are currently answering, or that the editor deduces must be answered. The current question is highlighted.

The center pane displays the interview questions. You answer the questions by using controls such as text boxes, radio buttons, or combo boxes located below each question. Whenever possible, the editor deduces answers from your system configuration and includes them in text boxes, combo boxes, and radio buttons. You can accept these answers or provide other answers.

The right pane displays important information about each question, such as:

- Background information
- Examples of answers
- Additional information about choosing an answer

- **Answer the questions in the configuration editor.**

  The following table presents the titles, answers, and information about each of the thirteen questions in the Demo TCK interview.

Question Title	Answer	Description
Welcome!		Briefly describes the purpose and function of the Demo TCK Configuration Editor.
Configuration Name	Demo_TCK	Names the interview file.
Description	tutorial	Describes the configuration.
How to Run Tests	On this computer	Runs both the JavaTest harness and the tests on the same computer.

Question Title	Answer	Description
Java Virtual Machine	The absolute path to the `java` command on a WIN32 system. For example: *jdk_inst_dir*`\bin\java.exe` or *jre_inst_dir*`\jre\java.exe`	Click the Browse button to activate a file chooser, or type the path directly in the text box.
Test Verboseness	Medium	Causes all executing tests to emit standard information messages.
Parameters...		Introduces the section of questions that collect information about which tests to run and how to run them.
Specify Tests to Run?	No	Runs all of the tests.
Specify an Exclude List?	No	Given No, specifies that an exclude list is not used for this test run.
Specify a Known Failures List	No	Given No, specifies that a list of known failures is not used in this test run.
Specify Status?	No	Specifies that prior run status is not used to filter the test run. Feel free to try it on subsequent runs.
Concurrency	1	Specifies the default concurrency setting (1).
Time Factor	1	Specifies the default standard time out value for each test (1).
Congratulations!		The configuration editor has collected all of the information it needs to run the tests. Click the Done button to save the interview. JavaTest interviews are saved to files that end with the `.jti` suffix. Use the file chooser to specify a file in a convenient location.

▼ Run Tests

1. **Set the view filter to Last Test Run.**

   Choose “Last Test Run” in the View Filter combo box located in the tool bar. This changes your “view” of the test tree so that you only see the results of the current test run. This is generally the view that most users prefer to begin with.

Note – Note that when you change to the Last Run filter before you do a test run, the folders and tests in the tree turn to gray, indicating that they are filtered out. This occurs because there are currently no results from a "last test run".

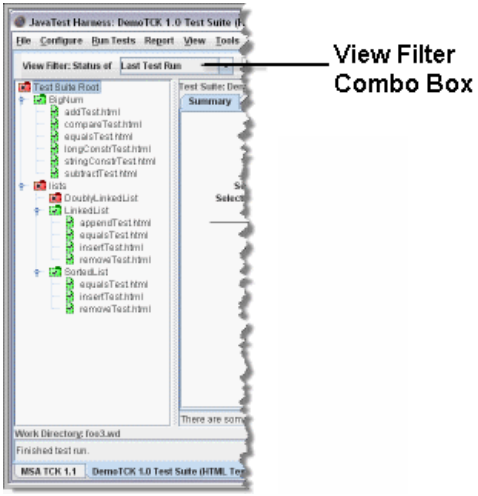
2. **Choose Run Tests > Start to start the test run.**

   The test suite should begin to run. You will see activity in the test tree panel that indicates which tests are running. You can also watch the progress of the test run in the progress monitor on the bottom-right portion of the JavaTest harness window and the pie chart in the Summary tab.

3. **Expand the test tree folders to reveal the tests.**

   Click on different test folders to expand the test tree.

FIGURE 2-3 Expanded Test Tree



As tests complete, the tests and their folders change color to represent their state. The following table briefly describes the colors and their meaning:

Color	Description
green	Passed
red	Failed
blue	Error — The test could not run properly. Usually indicates a configuration problem.
gray	Filtered out — Due to a parameter setting (for example, it is on an exclude list), the test is not selected to be run.
white	Not run

Folders reflect the state of the tests hierarchically beneath them. You know that the entire test suite passed if the test suite root folder is green. See the JavaTest online help for more information.

Note – The test `lists\DoublyLinkedList\InsertTest.java` intentionally contains errors and is supposed to fail as part of the tutorial. If any other tests fail, check your answers to the configuration interview.

Browse the Results

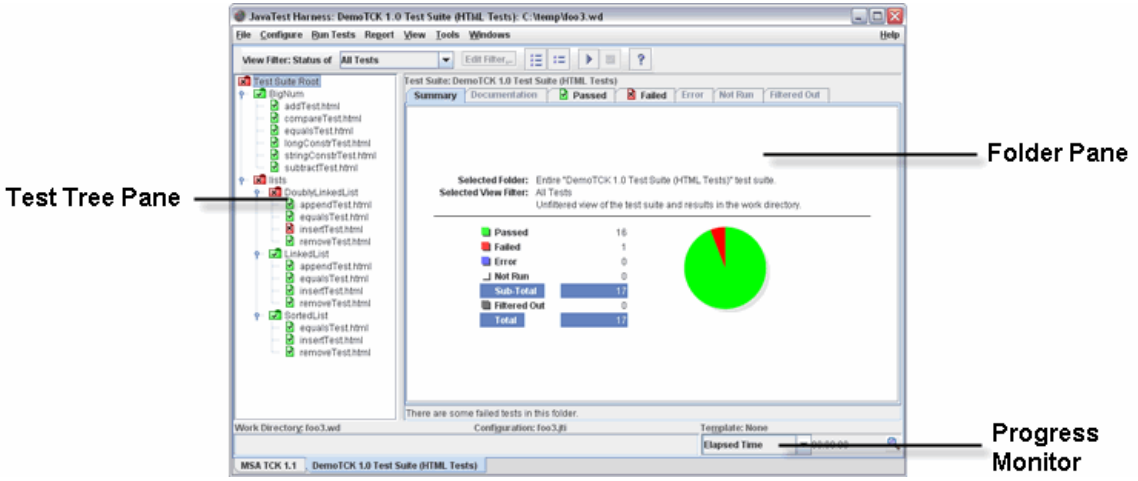
Now that the test run is complete, you will use the Folder tabbed pane and Test tabbed pane portion of the JavaTest harness to examine the results. You will also examine the output of the test that failed.

Note – The Folder tabbed pane and the Test tabbed pane occupy the same portion of the Test Manager window. The Folder tabbed pane is displayed when you choose a folder entry in the test tree and the Test tabbed pane is displayed when you choose a test entry in the test tree.

The Folder Pane

The Folder tabbed pane displays information about the tests in the selected folder.

FIGURE 2-4 The Folder Pane



▼ Browse the Folder Pane Results

1. **Click on the top folder in the test tree (the test tree root).**
2. **Click on the Summary tab (shown by default).**

   Notice the statistics displayed in the Summary panel. It describes how many tests in the test suite passed, failed, had errors, and were filtered out.

3. **Click on any of the other *folder* icons in the test tree.**

   Notice that the Summary panel changes to reflect the statistics for tests hierarchically beneath it.

4. **Click on the test tree root folder again.**

5. **Click on the Passed tab.**

   This pane contains a list of the tests that passed during the test run.

6. **Click on the Failed tab.**

   This pane contains a list of the tests that failed during the test run (only one test in this case).

7. **Double-click the `lists\DoublyLinkedList\InsertTest.java` test in the Failed tab.**

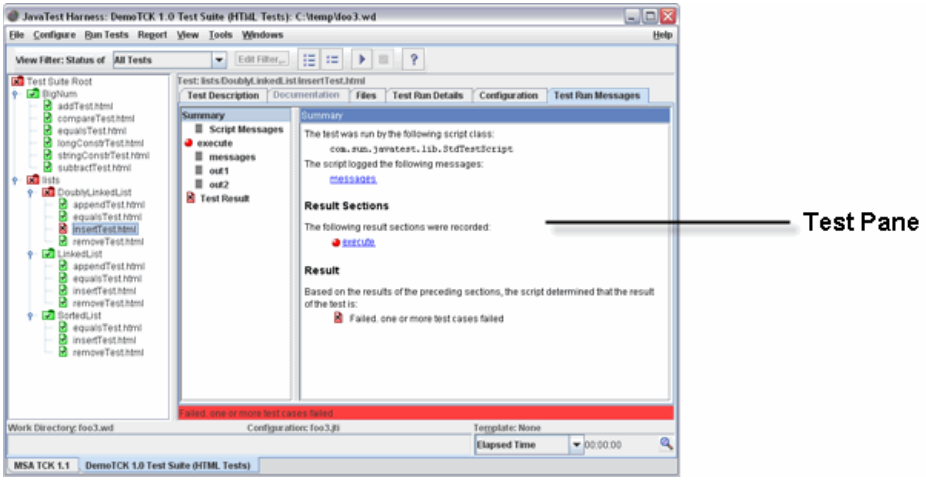
   This automatically selects the test in the test tree and changes the display from the Folder pane to the Test pane.

Note – To read more information about any of the panes, click on a tab to establish focus, and press F1 to activate online help about that pane.

The Test Pane

The Test tabbed pane displays information about the selected test. The five tabs provide information about the test and information about the results of its execution.

FIGURE 2-5 The Test Pane



▼ Browse the Test Pane Results

Click on the different tabs and examine the information the panes contain.

The following table briefly describes each tabbed pane:

Tab	Description
Test Run Messages	Displays messages generated during the selected test's execution
Test Run Details	A table of values generated during the selected test's execution
Configuration	A table of the configuration values used during the selected test's execution
Files	Displays the Java language source code and any other files related to the selected test
Test Description	A table of the test description values specified for the test

Note – To read more information about any of the panes, click on a tab to establish focus, and press F1 to activate the online help about that pane.

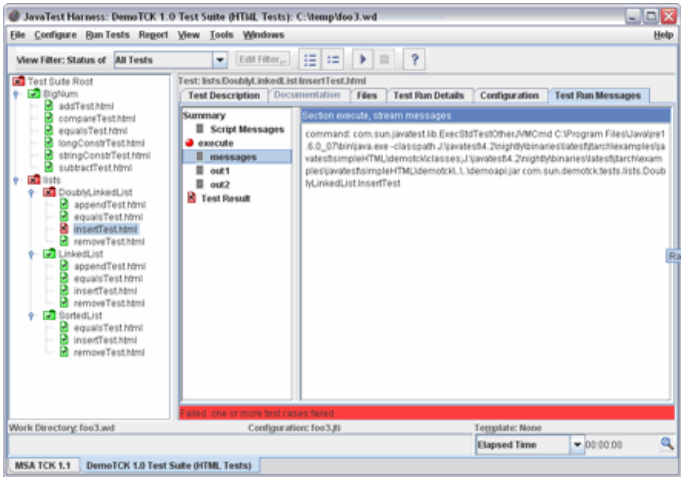
1. **Click on the Test Run Messages tab.**

   This pane provides access to any messages generated by the JavaTest harness or the test during execution. Notice that the various red icons indicate that the test failed.

2. **Click on the Execute/Messages entry in the left hand column.**

   The display on the right shows the command line used to run the test. Problems can often be debugged by examining how the test was invoked. In this case it was invoked correctly.

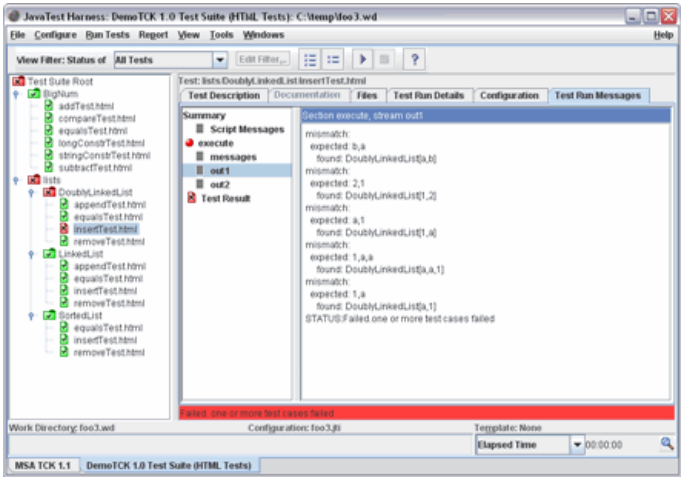
FIGURE 2-6 Test Messages



3. **Click on the out1 entry in the left-hand column.**

   The display on the right shows errors reported by the test. The messages indicate that either the test or the API contain errors — in this case the test contains errors.

FIGURE 2-7 Logged Error Messages



▼ Exclude the Failed Test

The JavaTest harness allows you to "exclude" tests from a test suite by specifying an *exclude list* file. This section shows you how to use the quick set mode of the configuration editor window to specify an exclude list that includes `lists\DoublyLinkedList\InsertTest.java`. Tests that are excluded are not executed during test runs, and though they are still displayed in the test tree, their status is not reflected in the pass/fail status of the test suite.

1. **Choose Configure > Change Configuration > Exclude List from the test manager menu bar.**

   The configuration editor window opens directly to a panel that allows you to specify an exclude list. This quick set mode allows you to quickly change values that change frequently between test runs. These values are also referred to as *standard values*. Note that standard values can also be changed using the configuration editor window in question mode.

2. **In the Exclude List pane, click Other.**

   This activates a tool with which you can specify a set of exclude lists.

3. **Click the Add button on the upper right portion of the tool.**

   This invokes a file chooser with which you can specify an exclude list. The current directory of the file chooser should be the directory in which you started the JavaTest harness. If it is not, please navigate to that directory.

4. **Double-click on the `lib` directory entry in the file chooser.**

5. **Choose the `demo.jtx` entry in the file chooser and click Select.**

   Notice that the exclude list (`demo.jtx`) is added to the Exclude Lists text box.

6. **Click Done in the configuration editor.**

7. **Change the view filter to "Current Configuration".**

   The Current Configuration filter shows which tests are selected and filtered out in the configuration, in effect a filter that shows which tests *will be* run next, as opposed to the Last Test Run filter which shows the tests that *were* run.

   Notice that the icon for the for `InsertTest.java` entry in the Test tree changes from red to gray. This indicates that the test has been filtered out and will not be executed. Also notice that the Test Suite Root folder has changed from red to green, indicating that all the currently selected tests have passed.

▼ Generate a Report

You can use the JavaTest harness to generate an HTML report that describes the results of the test run. All of the information contained in the report is available from the GUI; however, the following steps describe how to generate and browse a report that describes the test run done in the previous sections of this tutorial.

1. **Choose Report > Create Report**

   The Create a New Report dialog box opens.

2. **Specify the directory in which you want the report files to be written**

   If you wish to use a file chooser to specify the directory, click on the Browse button.

3. **Click the Create Report(s) button**

   The reports are generated and you are asked whether you want to view the report.

4. **Click Yes**

   The reports are displayed in the JavaTest report browser window. Scroll through the report and follow the various links to view data about the test run.

Note – If you wish to print the report, you can open the report in your favorite web browser and print it from there.

CHAPTER 3

Overview

JavaTest test suites are comprised of a number of components, many of which you, as the test suite architect, provide. This chapter introduces you to these components and some underlying concepts that are discussed in much greater detail later in this manual.

Test Suite Components

The most fundamental components of a test suite are the tests themselves. Tests are typically Java programs that exercise aspects of an API or compiler. To work well with the JavaTest harness, these files are organized in the file system hierarchically. The JavaTest harness finds the tests and displays them in the JavaTest GUI test tree based on this hierarchy.

Before the JavaTest harness can execute a test, it must know some fundamental things about the test — for example, where to find the class file that implements the test and what arguments the test takes. This information is contained in a *test description*. The test description is a group of name/value pairs that can be embodied in different ways — for example, as *tag test descriptions* and *HTML test descriptions*. Tag test descriptions are inserted directly into the test source files using Javadoc style tags. HTML test descriptions are HTML tables contained in HTML files separate from the test source and class files. The examples included with the JavaTest Architect's release demonstrate both types of test descriptions.

The JavaTest harness uses a specialized class called a *test finder* to locate tests, read test descriptions, and pass test description values to the JavaTest harness. As the architect, you specify a test finder that knows how to read the test descriptions you have designed for your test suite. The JavaTest Architect's release includes test finders that know how to read tag and HTML test descriptions; you can use the included test finders as-is, modify them, or create your own.

Once the test finder locates the test and reads the test description, it is up to the *test script* to actually run the test. The test script is a Java class whose job is to interpret the test description values, run the tests, and report the results back to the JavaTest harness. As the test suite architect, you are responsible for providing the test script that JavaTest uses. Test scripts can be very simple or complex, depending on the requirements of your test suite. A number of test script examples are included with the JavaTest Architect's release that you can use as is, extend, or use as a template for your test script.

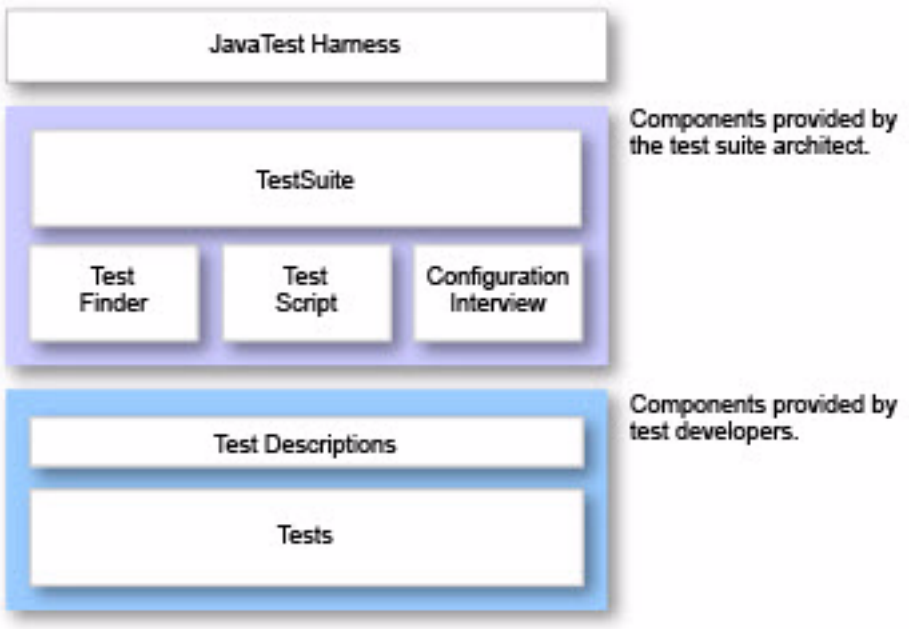
In addition to the fundamental information about each test, the test script might also require platform-specific information about each test to execute it. This information is provided by the person running the tests, usually by completing a wizard-like configuration interview designed by you. Platform-specific information includes information such as the location of the JVM to be used when running the tests, the names of remote computers, and other resources required to run the tests.

When test execution is complete, a test normally creates a `Status` object and passes it back to the test script; the test script then stores the test results in the `TestResult` object. Test status includes an integer that represents the status (pass, fail, error, not run) and a short message that describes what happened — for example, an error message. Test results include more detailed information about the results of the test's execution — for example, any additional messages produced by the test.

When the JavaTest harness loads a test suite, the first thing it does is read a file named `testsuite.jtt` located in the top-level directory of the test suite. The `testsuite.jtt` file is a registry of information about the test suite that includes the paths to the components described above and other static information about the test suite. The JavaTest harness internalizes this information in a `TestSuite` object that acts as a portal to all information about the test suite. Whenever the JavaTest harness requires information about the test suite, it queries the `TestSuite` object. As test suite architect, you create and maintain the `testsuite.jtt` file.

FIGURE 3-1 shows a graphical representation of the different test suite components:

FIGURE 3-1 Test Suite Components



The following table summarizes the sequence of steps the JavaTest harness uses to run test suites and your responsibility for each step. These steps are described in more detail in the following chapters.

TABLE 3-1 Summary of JavaTest Harness Operation

Events	Architect's Role
1 The user starts the JavaTest harness	Optionally, create a wrapper command to start the JavaTest harness in the right location and with the correct arguments.
2 The JavaTest harness reads `testsuite.jtt` to gather information about the test suite including the names and class paths for classes such as the finder, test script, and configuration interview. The JavaTest harness calls the `TestSuite` class, which in turn creates the `TestSuite` object. The JavaTest harness passes information from `testsuite.jtt` to the `TestSuite` class when it is created.	Determine what information is included in `testsuite.jtt` and what information (if any) is built directly into your test suite class. The components you create, including your test suite class are contained in a Java archive (JAR) file installed in the test suite. The path to the JAR file is specified in the `testsuite.jtt` file. Provide your test suite class
3 The JavaTest harness queries the `TestSuite` object to determine which test finder to use.	Provide your test finder class

TABLE 3-1 Summary of JavaTest Harness Operation *(Continued)*

Events	Architect's Role
4 The JavaTest harness starts the test finder. The test finder reads test descriptions and creates `TestDescription` objects. The JavaTest GUI displays the test tree.	
5 The user starts the test run. If the configuration information is incomplete, the JavaTest harness activates the configuration editor.	Provide the configuration interview
6 The JavaTest harness asks the `TestSuite` object to create a fresh copy of the test script for each test. The test script runs tests according to the information in the test description and environment. When the test is complete, the test script reports the test's exit status to the JavaTest harness.	Design the test execution model and create the test script
6 The test suite updates the `TestResult` object with the results of the test execution and writes the test results to a file in the work directory. Test results are described in "Test Result" on page 100.	
8 The JavaTest harness updates the GUI and/or displays information at the command line.	

Remote Execution

It is often convenient or necessary to run tests on a system other than the one running the JavaTest harness. In this case, an agent must be used to run the tests on the test platform and communicate with JavaTest harness. The JavaTest harness provides a general purpose agent (JavaTest Agent), but test architects can also create custom agents.

The JavaTest Agent is a lightweight program compatible with JDK 1.1 (does not require the Java SE platform, or Swing). The JavaTest Agent uses a bidirectional connection to communicate between the test platform and JavaTest—it supports both the TCP/IP and RS-232 protocols. Other types of connections can be added through the JavaTest API, for example, infrared, parallel, USB, firewire connections can be added and modelled on the existing system. If a test platform meets the following requirements the JavaTest Agent will probably work well:

- The device supports a communication layer that can last the duration of a test (couple of minutes)
- The agent code can be loaded into the device

If the test platform does not meet these requirements, the JavaTest API can be used to create a custom agent. Agents have been created to run tests on devices such as cell phones, PDAs, and pagers.

CHAPTER 4

Creating a Test Suite

This chapter leads you through the process of creating a very small working test suite quickly and easily by following step-by-step instructions. To simplify the process, conceptual information is generally not provided but is available in later chapters.

The test suite you create here can serve as the basis for your entire test suite. If your tests have no special requirements that the Standard Test Finder and Standard Test Script cannot accommodate, you may be able to create your product test suite by simply adding additional tests and creating a configuration interview to gather the information required to execute your tests.

Notes:

- The instructions in this chapter assume that you have completed the tutorial in Chapter 2 and that you have read Chapter 3.
- The instructions also assume that you are familiar with basic operating system commands on your system.
- In the examples, path names are given using the "\" file separator. If your system uses a different file separator, please substitute it where appropriate.

This chapter describes how to:

1. Create a test suite directory
2. Create a `testsuite.jtt` file
3. Copy `javatest.jar` to the test suite `lib` directory
4. Add appropriate classes to the `classes` directory
5. Create a test
6. Run the test suite

Other issues of interest regarding test suite creation are discussed at the end of the chapter.

Create a Test Suite

To create a test suite, follow the steps in these simple tasks.

▼ Create a Test Suite Directory

Create the directory and sub-directories for your test suite.

1. **Create the top-level test suite directory.**

   Create the directory somewhere convenient in your file system. This directory is referred to as *ts_dir* for the remainder of this chapter.

2. **Under *ts_dir*, create sub-directories named `tests`, `lib`, and `classes`.**

▼ Create the testsuite.jtt File

As described in Chapter 3, the JavaTest harness reads the `testsuite.jtt` file to find out information about your test suite. The following steps describe how to create the `testsuite.jtt` file for this test suite.

1. **Make *ts_dir* the current directory.**

2. **Create the** `testsuite.jtt` **file.**

   Enter the following information into a text editor:

```
# Test Suite properties file for DemoTCK test suite
# with tag-style tests
name=My Test Suite
id=1.0
finder=com.sun.javatest.finder.TagTestFinder
script=com.sun.javatest.lib.StdTestScript
interview=com.sun.javatest.interview.SimpleInterviewParameters
```

   You can substitute your own string values for the `name` and `id` properties.

Note – The `classpath` entry is not used here because the Standard Test Finder, Standard Test Script, and Simple Interview classes are all contained within `javatest.jar` which is automatically on the class path. If you create your own components, you must include the `classpath` entry to point to the JAR file that contains these classes. See Chapter 8 for more information about the `testsuite.jtt` file.

Save the file as *ts_dir*`\testsuite.jtt`.

▼ Copy `javatest.jar`

The test suite works best if there is a copy of the `javatest.jar` file in the `lib` directory of the test suite; this enables the JavaTest harness to automatically locate the test suite.

- **Copy** `javatest.jar` **from the** *jt_install*`\examples\javatest\simpleTags\demotck\lib` **directory to** *ts_dir*`\lib`**.**

Note – The `javatest.jar` file contains the `SimpleInterview` class that is used with this test suite (`com.sun.javatest.SimpleInterview.Parameters`). As your test suite becomes more complicated and customized, you may need to create a custom interview. See Chapter 6 for more information.

▼ Set Up the `classes` Directory

In order to execute, tests must have access to some of the classes contained in `javatest.jar`. Extracting these classes eliminates the need for each test to have `javatest.jar` on its class path. The the most convenient location to place these classes is the *ts_dir*`\classes` directory.

1. **Make** *ts_dir*`\classes` **the current directory.**
2. **Verify that the Java SE platform (version 1.5 or later) is in your path.**

   At a command prompt, enter:

   ```
   C:\> java -version
   ```

3. **From** `javatest.jar`**, extract the classes required to run the tests.**

   Use the following command line:

   ```
   jar -xvf ..\lib\javatest.jar com\sun\javatest\Test.class
   com\sun\javatest\Status.class
   ```

Note – As your test suite become more complex, you may have to add additional libraries to the `classes` directory.

▼ Use a Simple Test Template

The following instructions describe how to create a very simple test to add to your test suite. For more detailed instructions about writing TCK tests, see the *Test Suite Developers Guide*.

1. **Make** *ts_dir*`\tests` **the current directory.**

2. **Enter the test code into your favorite text editor.**

   The following template can be used as the basis for writing simple tests:

```
import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.Test;
/** @test
  * @executeClass MyTest
  * @sources MyTest.java
 **/
public class MyTest implements Test {
  public static void main(String[] args) {
      PrintWriter err = new PrintWriter(System.err, true);
      Test t = new MyTest();
      Status s = t.run(args, null, err);
      s.exit();
    }
  public Status run(String[] args, PrintWriter log1, PrintWriter log2) {
      Status result;
      // your test code here ...
      return result;
  }
}
```

Note that the section delimited with the `/** **/` characters is the test description portion of the test. It must be present for the JavaTest harness to locate and recognize the test. You will change all instances of **`MyTest`**, and replace the line **`// your test code here...`** with your own code. The following table describes the test description entries recognized by the Standard Test Script:

Test Description Entry	Description
test	Identifies the comment block as a test description and the containing file as a test
executeClass	Specifies the name of the test's executable class file (assumed to be located in the `classes` directory)
executeArgs	Specifies arguments (if any) that the test accepts
sources	Names the source files required to compile the test. This entry is required if you use the JavaTest harness to compile your tests. See Chapter 7 for more information. This tag is also used by the JavaTest harness to display a test's sources in the Files tab of the Test pane.
keywords	Specifies user-defined keywords that direct the JavaTest harness to include or exclude tests from a test run.

You can create simple tests by replacing the comment:

`// your test code here ...`

with code that tests your API. Note that the test must return a `Status` object as a `result`.

Note – You can find examples of simple tests at:
jt_install`\examples\javatest\simpleTags\demotck\tests`

▼ Create and Compile a Simple Test Example

The following sample is a *very* simple test you can use to get started.

1. **Save the following file as** `MyTest.java`**.**

   Be sure to copy the *entire* file, including the test description delimited with the `/** **/` characters

```
import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.Test;

/** @test
  * @executeClass MyTest
  * @sources MyTest.java
 **/

public class MyTest implements Test {
  public static void main(String[] args) {
      PrintWriter err = new PrintWriter(System.err, true);
      Test t = new MyTest();
      Status s = t.run(args, null, err);
      s.exit();
    }

  public Status run(String[] args, PrintWriter log1, PrintWriter log2) {
      Status result;
          if (1 + 1 == 2)
              result = Status.passed("OK");
          else
              result = Status.failed("Oops");
      return result;
  }
}
```

2. **Compile** `MyTest.java`**.**

   Use the following command on WIN32 systems:

```
C:\> javac -d ..\classes -classpath ..\classes MyTest.java
```

Use the following command on Solaris or Linux systems:

```
% javac -d ../classes -classpath ../classes MyTest.java
```

`MyTest.class` is created in the *ts_dir*`\classes` directory. As you add more and more tests you should organize them hierarchically in subdirectories.

Note – As you add more and more tests, you may want to use the JavaTest harness to compile the tests. For more information, see Chapter 7.

▼ Run a Test Suite

You are now ready to run the test suite.

1. **Make *ts_dir* the current directory.**
2. **Start the JavaTest harness.**

   At a command prompt enter:

   ```
   c:\> java -jar lib\javatest.jar -newdesktop
   ```

Note – The `-newdesktop` option is used here to ensure that the JavaTest harness starts up like it did in the tutorial — under normal circumstances you should *not* use this option. For information about JavaTest options, see the online help.

---

3. **Run the tests the same way you ran the tests in Chapter 2.**

   The configuration interview for this test suite contains a question not included in the tutorial configuration interview. Use the following information to answer the question:

Question Title	Answer	Description
Class Path	*ts_dir*`\classes`	The test uses library classes located in the `classes` directory. Click the Add button to activate a file chooser. Select the classes directory and click the Add File button.

Odds and Ends

This section takes a closer look at the components that make up a typical test suite and how they are organized. In addition, the various class paths required to run the JavaTest harness, the agent, and tests classes are discussed.

Note that much of the organization described here is optional; however, experience has shown that it works well for most test suites.

Top-Level Test Suite Directory

The top-level test suite directory generally contains the following files and directories:

TABLE 4-1 Top-Level Test Suite Files and Directories

File/Directory	Description
`testsuite.jtt`	A text file that serves as a registry of information about the test suite. This files includes the paths to plug-in components (for example, the test finder, test script, or configuration interview) as well as other static information about the test suite. The presence of this file defines the top-level directory of the test suite; therefore it *must* be located there. This file is described in detail in Chapter 8.
`lib\javatest.jar`	Contains all of the classes required to execute the JavaTest harness and library classes. The library classes can be used to simplify the creation of tests. If `javatest.jar` is located in the same directory as the `testsuite.jtt` file, or in the *ts_dir*`\lib` directory, the JavaTest harness automatically locates the test suite and does not prompt the user for the path to test suite directory. Note that it is very important *not* to put `javatest.jar` on the test suite class path. It is very large and scanning it for library classes at every test invocation impacts the performance of your test suite. The best option is to extract any referenced classes into the `classes` directory as shown in Step 3 in "Set Up the `classes` Directory" on page 28. Use of these library classes is described in Chapter 5.
`tests\`	Contains test source files and test descriptions. Tests should be organized hierarchically the way you want them to be displayed in the test tree. If you use the HTML test finder rather than the tag test finder, include the HTML test description files along with the test sources in the `tests` directory. For a discussion of test finders, see Chapter 9.

TABLE 4-1 Top-Level Test Suite Files and Directories *(Continued)*

File/Directory	Description
`classes\`	The directory that contains all of the compiled test classes and library classes required to run your tests. This directory is automatically placed on the JavaTest harness class path.
`lib\`	An optional directory that contains any other files required by your the test suite. These files might include: `jt`*testsuite*`.jar` — If you create a custom interview, or customize any of the JavaTest plug-in classes, you package the classes and interview files in a custom JAR file. See "The Test Suite JAR File" below for details. *test_suite_x.x*`.jtx` — The exclude list is a mechanism used by test suites to identify tests that should not be run.
`doc\`	An optional directory that contains documentation that describes how to run the test suite and specifies the rules for certifying a product.

The Test Suite JAR File

All of the components you create for the test suite should be delivered to the user in a single JAR file installed in the `lib` directory of the test suite. The JAR file is added to the class path in the `testsuite.jtt` file as described in Chapter 8. Experience has shown that it is best to organized the JAR file with the following directory structure:

```
com\
  your_company\
     your_product\
     Interview class files and resource files, More Info help
```

For example, the JAR file for the demo TCK test suite:

jt_install`\examples\javatest\simpleTags\demotck\jtdemotck.jar`

is organized like this:

```
com\
  sun\
     demotck\
     Interview class files and resource files, More Info help
```

If you provide a large number of components, you can further organize them into sub-packages:

```
com\
  your_company\
     your_product\
     Interview class files and resource files, More Info help
         lib\
         Everything else (TestSuite, Script, Finder, etc.)
```

Class Paths

When you create a test suite, it is important to keep in mind the three potential class paths that are involved:

- JavaTest class path
- Agent class path
- Test class path

Two of the ways described in the following sections in which you can set a class path are through a `CLASSPATH` environment setting or through a `-classpath` flag in the command line. The `CLASSPATH` environment setting is generally a safe way to set the classpath, although setting it as an environment variable is less explicit than using the `-classpath` flag.

The `-classpath` flag to the particular software development kit tool (such as `java` and `javac`) is generally the best way to set the class path if you know it explicitly. The main disadvantage of using the `-classpath` flag is that it overrides the `CLASSPATH` environment setting.

Note – If you have required classes that are set in the environment variable and you also use the `-classpath` flag, you must make special arrangements for the additional class paths to be set through the `-classpath` parameter.

Shell example:

```
% CLASSPATH=otherclasses javac -classpath classes -d out foo/
```

In this example, only *classes* set by `-classpath` are on the classpath. The *otherclasses* set by the `CLASSPATH` environment setting are dropped.

Shell example:

```
% CLASSPATH=otherclasses javac -classpath $CLASSPATH:classes d out foo/
```

In this example, the environment setting is added to the `-classpath` argument. The resulting classpath is *otherclasses*`:`*classes*.

JavaTest Class Path

This is the class path that the JavaTest harness uses to access its classes, libraries, and your plug-in classes. The JavaTest class path can be set by means of:

- The `CLASSPATH` environment variable
- The `-classpath` option to the Java runtime
- The `-jar` option to the Java runtime (this is the standard)

In addition, each test suite can use the `classpath` entry in the `testsuite.jtt` file to extend the class path. The `classpath` entry is used to add custom plug-in components and interviews that you create.

Agent Class Path

Often you must run tests on a system other than one on which the JavaTest harness runs. In this case you use an agent (such as the JavaTest Agent) to run the tests on that system. The agent class path is used by the agent to access its classes, libraries, and any plug-in classes. The class path can be set by means of:

- The `CLASSPATH` environment variable
- The `-classpath` option to the Java runtime
- The `-jar` option to the Java runtime
- Some other platform-specific mechanism

Test Class Path

This is the class path used by the tests during execution. It is normally the responsibility of the configuration interview and/or test script to set the class path for each test in the test environment command entry (see "Command Strings" on page 50). Test classes are normally located in the *ts_dir*`\classes` directory, you normally include this on the test class path. You can also put any classes that your tests require in *ts_dir*`\classes` and they will be found.

Note – If your platform requires that tests run in the same JVM as the agent, you must include the classes required by the tests on the agent class path. In this case your interview need not put a test class path in the test environment command entry.

CHAPTER 5

Writing Tests

This chapter describes how to write tests that work well with the JavaTest harness. Special attention is paid to the test execution model implemented by the Standard Test Script which is designed to work well with test suites that test the compatibility of Java APIs and should work well with most Java SE technology-based TCK test suites.

Note that this discussion focuses on the mechanics of writing tests that work with the JavaTest harness. For information about the "art" of writing compatibility tests, see the *Test Suite Developers Guide*.

The example test suites included with the JavaTest Architect's release contain numerous test examples. See the following directories:

jt_install`\examples\javatest\simpleTags\tests`

jt_install`\examples\javatest\simpleHTML\tests`

You might find it helpful to refer to those tests as you read this chapter.

The Test Execution Model

The design and invocation of a test is a reflection of the test execution model that you design for your test suite. The test execution model describes the steps involved in executing the tests in your test suite and is implemented by the test script.

As you design your test suite you should think about how your tests are going to be executed. Some typical questions you might ask:

- Is each test invoked by executing a single class?
- Do the tests require multiple steps, implemented by multiple class invocations?
- Must test classes be started on different machines and in a specific order?

Most TCK test suites test specific aspects of an API. These types of tests lend themselves to an execution model in which tests are run by invoking a single class that exercises a method or group of methods. The JavaTest Architect's release includes the Standard Test Script (`com.sun.javatest.lib.StdTestScript`) that implements this test execution model. The Standard Test Script is discussed in more detail in Chapter 10.

If your test suite requires a more complex test execution model, you have to create a test script to implement it. See Chapter 10 for information about creating a custom test script.

Note – The test execution model implemented by the Standard Test Script includes an optional compilation step. The Standard Test Script can be used to:
- Execute precompiled tests
- Compile the tests
- Compile and execute the tests

See Chapter 7 for more information about compiling tests with the JavaTest harness.

The `Test` Interface

If you plan to run your tests using the execution model embodied by the Standard Test Script, the tests must implement the `run` method of the interface `com.sun.javatest.Test`. The `Test` interface provides a very flexible mechanism that is well suited for creating most tests. If the `Test` interface does not suite your needs, you can write your own interface. You can find information about creating your own interface in Chapter 10.

The `Test` interface `run` method takes an array of strings and two output streams and returns a `Status` object. The array of strings is taken from the `executeArgs` entry in the test description. The output streams are provided by the JavaTest harness; any output written to the output streams is saved in the `TestResult` object and is displayed in the Test Run Messages tab in the JavaTest GUI. The end result of the test is a `Status` object — a combination of an integer code and a message string (see "Test Status" on page 40).

The following code example shows a template for tests written to work with the Standard Test Script; the areas you change are in bold font:

```
import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.Test;
```

```
/** @test
  * @executeClass MyTest
  * @sources MyTest.java
 **/

public class MyTest implements Test {
  public static void main(String[] args) {
      PrintWriter out = new PrintWriter(System.err, true);
      Test t = new MyTest();
      Status s = t.run(args, out, null);
      s.exit();
     }

  public Status run(String[] args, PrintWriter out1, PrintWriter
out2) {
      Status result;
      // your test code here ...
      return result;
  }
}
```

Note that the section delimited with the `/** **/` characters is the test description portion of the test which is described in more detail later in this chapter in "Test Description Entries" on page 41. The `Status` object is described in "Test Status" on page 40.

Class Paths

The `com.sun.javatest.Test` interface is delivered in `javatest.jar`; however, you should extract it into your test suite's `classes` directory so that it is easily available to all of your test classes.

Note – To improve test performance, never add `javatest.jar` to test paths anywhere in your test suite. If you use classes provided in `javatest.jar`, extract them into your test suite's `classes` directory.

Test Status

The `Status` object is an integer/string pair that encodes the exit status of the test. The JavaTest harness supports the following exit status values:

TABLE 5-1 Exit Status Values

Status	Meaning
PASSED	A test passes when the functionality being tested behaves as expected.
FAILED	A test fails when the functionality being tested does not behave as expected.
ERROR	A test is considered to be in error when something (usually a configuration problem) prevents the test from executing as expected. Errors often indicate a systemic problem — a single configuration problem can cause many tests to fail. For example, if the path to the Java runtime is configured incorrectly, no tests can run and all are in error.

Note – The NOT_RUN status indicates that the test has not been run. This is a special case and is reserved for internal JavaTest harness use only.

The integer portion of the `Status` object represents the exit status; the string portion is a message that summarizes the outcome (for example, an error message). Only the short integer portion is used by the JavaTest harness to determine the test status. The message string provides information to the user running the test. The message is passed to the test script which writes it into the test result file.

Note that the object is immutable once it is created — if the test script modifies the message string it must take the `Status` object created by the test and recreate the `Status` object including the new message string.

The JavaTest harness uses the information in the `Status` object in its GUI status displays and reports.

There are two important methods in the `Status` API that your tests can use: `passed()` and `failed()`. Both methods take a string as an argument and return a `Status` object. The JavaTest harness displays these strings in the Test Run Message tab in the JavaTest GUI and they can be an important source of information to users running the tests. The following example shows how these methods are used:

```
public Status run(String[] args, PrintWriter out1, PrintWriter out2) {
  Status result;
     if (1 + 1 == 2)
        result = Status.passed("OK");
```

```
      else
         result = Status.failed("Simple addition performed incorrectly");
  return result;
  }
}
```

The test entries in the reports generated by the JavaTest harness are grouped based on the string arguments you supply to `Status.passed` and `Status.failed`. It's generally a good idea to keep all of the `Status.passed` messages short and consistent so that similar tests are grouped together in reports. `Status.failed` messages should generally be longer and more descriptive to help the user determine why the test failed. Complete details should be written to the output stream.

See the API documentation (`doc\javatest\api`) for the `Status` class.

Test Description Entries

All tests must have an associated test description that contains entries that identify it as a test and provide the information required to run it. Test descriptions are located and read by a test finder; the two standard test finders included with the JavaTest harness read two styles of test description: tag test descriptions and HTML test descriptions. It is your decision as test suite architect which style to use (you can even create a custom style). Test finders are discussed in detail in Chapter 9. For simplicity, only the tag style is shown in this chapter.

Test finders read all entries listed in the test description and add them to the `TestDescription` object. The Standard Test Script looks for and uses the values specified in the `executeClass`, `executeArgs`, and `sources` entries; the script disregards any other entries. You can create your own custom script that recognizes additional test description entries and validate those entries. See Chapter 10 for more information.

The following table describes the entries understood by the Standard Test Script:

TABLE 5-2 Default Test Description Entries

Test Description Entry	Description
test	Identifies the comment block as a test description. This entry is required. There is no "test" entry in the `TestDescription` object.
executeClass	Specifies the name of the test's executable class file (assumed to be located in the `classes` directory). This entry is required.
executeArgs	Specifies the arguments (if any) that the test accepts. This entry is a list of strings separated by white space. This entry is optional.
sources	Specifies the names of the source files required to compile the test. This entry is required if you use the JavaTest harness to compile your tests. See Chapter 7 for more information. This tag is also used by the JavaTest harness to display a test's sources in the Files tab of the Test pane. This entry is optional.
keywords	Specifies keywords that the user can specify to direct the JavaTest harness to include or exclude tests from a test run. Keyword values consists of a list of words (letters and numbers only) separated by white space. This entry is optional.

The following code snippet shows how a tag test description appears in a test source file:

```
/** @test
  * @executeClass MyTest
  * @sources MyTest.java
  * @executeArgs arg1 arg2
  * @keywords keyword1 keyword2
 **/
```

Keywords

You can add keywords to test descriptions that provide a convenient means by which users can choose to execute or exclude pre-selected groups of tests. The person who runs the test suite can specify keyword expressions in the configuration editor. When the test suite is run, the JavaTest harness evaluates the keyword expressions and determines which tests to run based on the keywords specified in the test description. See the JavaTest harness online help for information about specifying keyword expressions.

Multiple Tests in a Single Test File

If you find that you are writing lots of very small tests to test similar aspects of your API, you can group these similar tests together as *test cases* in a single test file. Tests that contain test cases should use the `com.sun.javatest.lib.MultiTest` class rather than the `com.sun.javatest.Test` class. `MultiTest` extends `com.sun.javatest.Test` to add this functionality. One of the major benefits of using `MultiTest` to implement test cases, is the test cases can be addressed individually in the test suite's exclude list. Another advantage to using MultiTest is that the test cases are run in the same JVM which is generally faster than starting a new JVM for each test. The downside to using MultiTest is that tests are more susceptible to errors introduced by memory leaks.

`MultiTest` is included with the JavaTest release as a standard library class. `MultiTest` is a class that implements the `com.sun.javatest.Test` interface and allows you to write individual test cases as methods with a specific signature. These methods cannot take any parameters and must return a `com.sun.javatest.Status` object as a result. Argument decoding must be done once by a test for its test case methods. `MultiTest` uses reflection to determine the complete set of methods that match the specific signature. `MultiTest` calls test case methods individually, omitting any tests cases that are excluded. The individual `Status` results from those methods are combined by `MultiTest` into an aggregate `Status` object. The test result is presented as a summary of all the test cases in the test.

The following example shows a very simple test that uses MultiTest to implement test cases:

```
import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.Test;
import com.sun.javatest.lib.MultiTest;

/** @test
  * @executeClass MyTest
  * @sources MyTest.java
 **/

public class MyTest extends MultiTest {
  public static void main(String[] args) {
      PrintWriter err = new PrintWriter(System.err, true);
      Test t = new MyTest();
      Status s = t.run(args, null, err);
      // Run calls the individual testXXX methods and
      // returns an aggregate result.
```

```
        s.exit();
      }
    public Status testCase1() {
        if (1 + 1 == 2)
            return Status.passed("OK");
        else
            return Status.failed("1 + 1 did not make 2");
    }
    public Status testCase2() {
        if (2 + 2 == 4)
            return Status.passed("OK");
        else
            return Status.failed("2 + 2 did not make 4");
    }
    public Status testCase3() {
        if (3 + 3 == 6)
            return Status.passed("OK");
        else
            return Status.failed("3 + 3 did not make 6");
    }
}
```

For more information about `com.sun.javatest.lib.MultiTest`, please refer to the API documentation.

Subtyping MultiTest

If you create a number of tests that are similar you can create a super class to implement functionality they have in common. You can also create this class as a subtype of the `MultiTest` class rather than the `Test` interface so that you can take advantage of the test case functionality it provides. Such subtypes are typically used to perform common argument decoding and validation, or common set-up and tear-down before each test or test case.

Organizing Tests Within Your Test Suite

This section describes some guidelines about how to organize your test source and class files.

Source Files

It is very important to ship the source files for tests in your test suite. Test users must be able to look at the sources to help debug their test runs.

Test sources should be located with the files that contain their test descriptions. If you use tag test descriptions, the test description is included as part of the source file; however, if you use HTML test descriptions, they are contained in separate HTML files that should be included in the same directories as their test source files.

The JavaTest harness assumes that tests are organized hierarchically in a tree structure under the *ts_dir*`/tests` directory. The test hierarchy contained in the `tests` directory is reflected in the test tree panel in the JavaTest GUI (technically, it is a tree of the test descriptions). When you organize your `tests` directory, think about how it will look in the test tree panel. In test suites that test APIs, the upper part of the tree generally reflects the package structure of the product you are testing. Farther down the tree, you can organize the tests based on the sub-packages and classes being tested. The leaves of the tree might contain one test per method of that class. In some cases it might make sense to organize the tree hierarchy based on behavior; for example, you could group all event handling tests in one directory.

Class Files

Experience has shown that it is a good idea to place all of your test class files in the *ts_dir*`\classes` directory rather than locating them with the source files in the *ts_dir*`\tests` directory. Placing class files in the `classes` directory has the following benefits:

- It simplifies the specification of the test execution class path, especially on smaller devices that can only specify a single class path for all the tests.
- The standard configuration interview automatically places *ts_dir*`\classes` on the test class path
- It permits easier code sharing among tests

Note – In some cases the test platform may dictate where you can put your classes. For example, if your test platform requires the use of an application manager, it may require that your classes be placed in a specific location.

Error Messages

It is important that your tests provide error messages that test users can readily use to debug problems in their test runs. One useful method is for your error messages to compare expected behavior to the actual behavior. For example:

```
Addition test failed: expected a result of "2"; got "3"
```

Longer detailed messages should go to the test and/or test script diagnostic streams. Use the Status object for shorter summary messages.

CHAPTER 6

Creating a Configuration Interview

As you design your test suite, you must decide how to provide the JavaTest harness with all of the information required to execute your tests. Some of this information is static — it is known prior to runtime through the test description mechanism. However, some information cannot be determined ahead of time and differs based on the context in which the tests are run. This information is called the *configuration* and is obtained from the user through a configuration interview that you design. The configuration interview is presented to the user in the JavaTest configuration editor and consists of a series of simple questions that the user answers. The interview exports the answers in a format called a *test environment* that the JavaTest harness understands.

This chapter describes how to create and package a configuration interview.

Designing Your Configuration

This section focuses on the design of the configuration information and how to determine what information is necessary to run your tests suite.

What is a Configuration?

The configuration provides the JavaTest harness with the information it needs to execute tests. This information falls in the following categories:

- Information required by the script to execute the tests
- Information required by tests. This information augments the test description and usually consists of information that changes based on the test execution context (for example, the platform or network).
- Information that determines which tests to include or exclude from a test run

These categories are discussed in the following sections.

Test Script Information

A test script is responsible for running your tests. The test script knows the series of steps required to execute each test. It typically relies on *test commands* to perform each step and you design your configuration to provide the test commands (and their arguments) that the test script uses to execute each test. Test commands are Java classes that the test script instantiates to run tests.

As an example, the Standard Test Script uses a single step to execute tests; that step is defined in the configuration entry called `command.execute`. The configuration interview is responsible for setting the value of `command.execute` so that the Standard Test Script uses the appropriate command and arguments. For example, you can tell the Standard Test Script to use the `ExecStdTestOtherJVMCmd` command which executes tests in a process on the same computer that runs the JavaTest harness:

`command.execute=com.sun.javatest.lib.ExecStdTestOtherJVMCmd` *args*

If you intend to execute the tests differently; for example, on a different computer, you would define `command.execute` differently in your configuration. For a list of test commands included with the JavaTest release, see Appendix A. For information about creating custom test commands, see Chapter 10.

Test Description Entries

In the previous chapters of this manual, you have seen that most test descriptions are static; these entries consist of values that are known ahead of time and can be specified directly. In some cases these arguments cannot be determined ahead of time, especially test arguments (`executeArgs`). For example, tests that test network APIs may require the names of hosts on the network to exercise the API. If the test suite runs in different locations and on different networks, these values cannot be known ahead of time by the test developer. The configuration interview is expected to collect this information and make it available to the test.

A script may allow the test developer to specify variables in some test description entries that are defined in the configuration; these variables are prefixed with the "`$`" character. For example the Standard Test Script allows variables in the `executeArg` entry; in the case of a network test, here is what the test description might look like:

```
/** @test
  * @executeClass MyNetworkTest
```

```
   * @sources MyNetworkTest.java
   *@executeArgs -host $testHost -port $testPort
  **/
```

The arguments to the `executeClass` and `sources` entries are static — they are known ahead of time and do not change based on the context in which the test runs. The host names or IP addresses cannot be known ahead of time and are specified as variables to which the JavaTest harness assigns values when the test is run. The test suite's configuration interview asks the user to specify the values of the hosts and port numbers required to run the test; the values of `$testHost` and `$testPort` are defined from those answers. The configuration interview creates entries in the test environment as name/value pairs. For example:

```
testHost=129.42.1.50
testPort=8080
```

Which Tests to Run

The JavaTest harness provides a number of ways that the user can specify which tests in the test suite to run. These *standard values* can be specified by the user in the configuration editor window question mode or quick set mode. You can easily include interview questions that gather this information at the end of the interview for you and require no extra work on your part.

Designing Your Interview

The goal of the configuration interview is to create (or *export*) a test environment. The test environment consists of one or more command templates that the test script uses to execute tests and the set of name/value pairs that define values required to run the tests.

The previous section described how to think about the kinds of configuration values your test suite needs; this section focuses on how you collect configuration values and translate them into test environment entries.

Command Strings

The most complex test environment entries are almost always the command strings the test script uses to execute the tests. A command string is a template that specifies the command used by the test script to execute the test. A command string contains symbolic values (variables) whose values are provided when the test is executed.

The complexity of these entries is determined by the versatility required by the test suite. If the test suite is always run on the same computer, in the same network, the command string is probably very easy to specify. In many cases the computing environment varies considerably, in which case the command strings are built up largely from answers that users provide in the configuration interview.

As previously described, test scripts depend on test commands to know how to execute tests in specific ways. The JavaTest release contains a set of standard library test commands that you can use to execute tests. The following table describes the most commonly used test commands. These test commands are described in more detail in Appendix A.

TABLE 6-1 Commonly Used Test Commands

Test Command	Description
`ExecStdTestSameJVMCmd`	Executes a simple API test in the same JVM as the caller. Typically used with the JavaTest Agent.
`ActiveAgentCommand` `PassiveAgentCommand`	Execute a subcommand on a JavaTest Agent running in active or passive mode

If your platform requires a custom agent in order to run tests, use the test command designed for use with that agent.

Commands and command templates are described in more detail in Chapter 10.

The examples in this section show how to create command entries for the Standard Test Script using two of these commands: `ActiveAgentCommand` and `ExecStdTestOtherJVMCmd`.

Example 1

The Standard Test Script uses the value of the command entry `command.execute` to execute tests. If the tests are executed on the same computer running the JavaTest harness, a typical command entry for the Standard Test Script looks something like the following:

```
command.execute=com.sun.javatest.lib.ExecStdTestOtherJVMCmd
   C:\JDK\bin\java.exe -classpath $testSuiteRootDir\classes
   $testExecuteClass $testExecuteArgs
```

The portion of the entry to the left of the "=" is the name of the test environment entry, the portion to the right is the command string.

Let's examine the command string in detail:

`com.sun.javatest.lib.ExecStdTestOtherJVMCmd`

The first part of the command string is the name of the test command class used to execute the test classes. In this example the command executes tests in a process on the same computer that runs the JavaTest harness.

Interview implications:

Your configuration interview specifies the command to use to execute the tests. If the API you are testing always runs in a known computing environment, your interview might create this part of the entry without input from the user. However, if the API being tested can be run in different ways, you must ask the user which way they choose to run it and provide the appropriate test command based on the user's input.

Imagine an API that can be tested on the same computer running the JavaTest harness, or on a different computer on the same network. In this case the interview must determine which way the user intends to run the tests and provide the appropriate command — `ActiveAgentCommand` or `ExecStdTestOtherJVMCmd`.

`-classpath` *ts_dir*`\classes`

The class path required by the tests. Replace *ts_dir* with the path to your test suite. To enhance performance, you should place all library classes required to run the test classes in the `classes` directory.

See "Test Environment Variables" on page 53 for a list of available variables.

Interview implications:

You can determine the path to your test suite inside your interview. See "Exporting the Test Environment" on page 57 for details. If the test classes require no additional classes be on the class path other than the ones you provide in the test suite's `classes` directory, your interview can insert the class path value directly into the

entry without asking the user for input. If additional class path entries may be required, your interview may include questions that ask the user to provide additional entries that your interview appends to the class path.

This environment entry that can get more complicated if the test suite may be run using different versions of the Java runtime. Some Java runtime systems do not use the `-classpath` option; for example, they might use a syntax such as `-cp` or `/cp`. Additionally, some systems use the "`:`" character as the class path separator and others use the "`;`" character. If this is the case, your interview must include additional questions that determine the platform on which the tests are run so that you can create the appropriate command entry.

`C:\`*JDK*`\bin\java.exe`

The path to the Java runtime command used to execute the test classes.

Interview implications:

This path almost certainly differs from user to user, so almost any interview must obtain this path from the user. The interview libraries include a question type named "file" that is very useful for obtaining path names.

Although no additional options or arguments are shown in this example, many Java runtimes or test suites require additional options and arguments. If your tests require any additional options, you include them in additional portions of the entry.

`$testExecuteClass`

A variable that represents the name of the test class. The test script obtains the class name from the `executeClass` entry in the test description and provides it at runtime.

Interview implications:

The interview adds the variable to the environment entry.

`$testExecuteArgs`

A variable that represents the arguments specified in the test description. The test script obtains this value from the test description and provides it at runtime.

Interview implications:

The interview adds the variable to the environment entry.

Example 2

For this example, imagine a test suite that runs in a limited environment — it always runs on a remote system using the JavaTest Agent in passive mode. The command entry looks like this:

```
command.execute=com.sun.javatest.lib.PassiveAgentCommand
    -host myHost -port 501
    com.sun.javatest.lib.ExecStdTestSameJVMCmd
    $testExecuteClass $testExecuteArgs
```

Although this command is quite long, because of its limitations most of it is boilerplate; the only values that your interview has to gather from the user are the arguments to the `-host` and `-port` options.

Test Environment Variables

The following variables are available for use in test descriptions if you use the Standard Test Script or a test script derived from it. If you create a custom test script, it can provide additional values.

TABLE 6-2 Test Environment Variables

Variable Name	Definition
`$testExecuteArgs`	The value for the `executeArgs` parameter from the test description of the test being run
`$testExecuteClass`	The value of the `executeClass` parameter from the test description of the test being run
`$testSource`	The value of the `source` parameter defined in the test description of the test being run. Valid only when using the JavaTest harness to compile a test suite. See Chapter 7.

Writing Your Interview

The previous two sections focused on the design of your configuration and your interview; this section focuses on writing the code to implement the interview.

This section takes a high-level view of the process of writing configuration interviews; complete, working code examples are provided separately from this manual. These examples are:

Demo TCK interview

The Demo TCK is a simple test suite created to demonstrate the basic principles of writing and running test suites. The Demo TCK was featured in Chapter 7. The source code and More Info files for the configuration interview used in the Demo TCK test suite are included in the JavaTest Architect's release at the following location:

jt_install`\examples\javatest\simpleTags\src`

Demo Interview

The Demo Interview is a self-documenting JavaTest interview that demonstrates all of the interview question types, and other important interview techniques. A special viewer allows you to view the source of a question as you run it. Follow these instructions to start the Demo Interview:

▼ Start the Demo Interview

1. **In a command window make the following your current directory:**

   jt_install`\examples\javatest\interviewDemo\demotck`

2. **Start the Demo Interview test suite**

   At the command prompt enter:

   ```
   C:\>java -jar lib\javatest.jar -newDesktop
   ```

   The `-newdesktop` option is used here to ensure that the JavaTest harness loads the correct test suite. For information about JavaTest options, see the online help.

3. **Choose Configure > New Configuration to start the interview**

   Follow the directions in the interview. You can also browse the source for the interview at:

   jt_install`\examples\javatest\interviewDemo\src`

Interview Classes

Interviews are built from the following classes:

`com.sun.javatest.InterviewParameters`

The top-level class used to build JavaTest configuration interviews. This class is a special subtype of `com.sun.interview.Interview`
that provides the API required by the JavaTest harness. You do not normally use this class directly, see `BasicInterviewParameters` below.

`com.sun.interview.Question` `(and its subtypes)`

Questions are the primary constituent elements of interviews. Questions provide text and appropriate controls and serve as a place to store the user's response.

`com.sun.interview.Interview`

The base interview class. This class is used directly to implement sub-interviews (if any).

`com.sun.javatest.interview.BasicInterviewParameters`

A subtype of `com.sun.javatest.InterviewParameters` that provides standard questions for all of the "standard" configuration values (for example, which tests to execute). You usually subtype this interview and expand it to obtain your specific test environment information. The `BasicInterviewParameters` class is flexible, see "Putting it All Together" on page 62 for details.

For more information about these classes, please refer to the API documentation available in `doc\javatest\api`.

To create a configuration interview, you normally provide a subclass of the `BasicInterviewParameters` class and add questions to the interview. This class is responsible for collecting all test environment and standard value information and providing it to the JavaTest harness.

Interviews can contain nested sub-interviews. The choice of whether to break interviews into smaller sub-interviews is a design decision based on manageability — generally interviews over 20 questions are candidates for this kind of hierarchical organization. Interviews often contain a number of branches, and these branches are also often good candidates for becoming sub-interviews. Sub-interviews directly extend `com.sun.interview.Interview`.

The Current Interview Path

As mentioned in the previous section, interviews are often composed from sub-interviews that branch off of the main interview. During the interview process, branches of the interview can become inactive because the user changes the answer to a question; the branch can become reactivated if the user later changes the answer back. When a user completes a configuration interview, the answers to all questions the user has ever answered are stored on disk in an interview data file with the `.jti` extension. Because active and inactive questions are present in the interview data

file, whenever the JavaTest harness needs configuration information (for example, to run tests or to display the environment) the JavaTest harness must determine the *current interview path*.

To determine the current interview path, the JavaTest harness starts at the first question and queries each question for the next question on the path, attempting to reach the Final question (see TABLE 6-3TABLE 6-3for a description of different question types). If it does not reach the Final question, the interview is considered incomplete; the test configuration cannot be exported and the test suite cannot be run until the missing questions are answered. If the user attempts to run the test suite with an incomplete interview, they are asked whether they want to complete the interview at that time — if they do, the configuration editor is activated.

Determining the Next Question

Every question except the Final question must provide a `getNext()` method that determines the next (successor) question. The successor question can be fixed (constant) or determined based on the answer of a current question or on the cumulative answers of multiple preceding questions. Questions can also provide no successor question (by returning `null`). Lack of a successor question usually means that the current question is unanswered or contains an error; in that case the interview is incomplete.

You may add questions to the interview that gather no configuration information, they are only used to determine the next question in the interview. These are typically Choice questions used to determine a branch point in the interview. For example, you might include a question that asks the user whether they want to execute the tests locally (on the computer running the JavaTest harness) or on a remote computer using the JavaTest agent. Depending on the answer, you branch to the questions that gather information about how to run the JavaTest Agent.

Error Checking

If the user provides an invalid answer to a question, the interview cannot proceed. You use the boolean `isValueValid()` method to check the validity of an answer before you proceed to the getNext() method. You can handle error conditions in two ways: by returning `null` which causes the configuration editor to display the "Invalid response" message in red at the bottom of the question pane, or by making the successor question an Error question that causes the configuration editor to display a pop-up window with an error message that you provide (see `ErrorQuestion` in).

Generally, an "Invalid response" message is sufficient if the error is a simple one; for example, if the user answers an integer question with a letter. However, for more subtle errors (for example, if an answer conflicts with a previous answer), it is necessary to provide more information to the user in a pop-up window.

Exporting the Test Environment

As previously mentioned, one of the goals of the interview is to produce a test environment. The JavaTest harness uses the `InterviewParameters` class's `getEnv()` method to obtain the test environment.

If you extend `BasicInterviewParameters` to create your interview, it provides an implementation of the `getEnv()` method that uses the values you export.

If, however, you extend `InterviewParameters` directly, you must provide a `getEnv()` method that gathers answers from the main interview and any sub-interviews and returns an `TestEnvironment` object. The best and simplest way to implement the `getEnv()` method is to use the interview's `export()` method, which in turn calls the `export()` method of each question on the current interview path that provides one. Note that an interview does not normally override/provide `export()`— it is provided automatically. When it is time to export the test environment, the `getEnv()` method calls `export()` to gather their test environment information. These questions export their values into a `Map` object from which you can construct a test environment. For detailed examples see the source code examples in the *jt_install*`\examples` directory.

When exporting the test environment, you can use the `getTestSuite()` method to get information about the test suite. This information (for example, the location of the test suite) is often useful in building test environment entries.

Note – It is generally a very good idea for the controlling question to precede the questions that collect a given value, because the question text can provide information to the user about the series of questions coming up.

Question Types

The `Question` class is a base class that provides the different types of questions that you use to build your interview. You need not concern yourself about the GUI layout of a question; the configuration editor automatically determines the presentation of each question based on the question's type.

The following table lists all of the question types and shows (when applicable) how they are represented in the configuration editor.

TABLE 6-3 Question Types

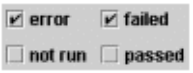
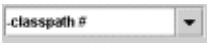
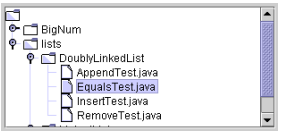
Question Type	Description	GUI	Example
ChoiceArray	A set of independent boolean choices	Set of named checkboxes	
Choice	A set of alternative choices	Combo box or radio buttons, depending on the number of choices	
Error	A pseudo question used to present error messages	Pop-up dialog box	
File	A single file	Type-in field with associated file chooser	
FileList	A set of files	A list box with an associated file chooser	
Final	1. A pseudo question that marks successful completion of the interview	Text only, no user input	
	2. A pseudo question that marks the end of a sub-interview	For internal use only; never displayed	
Float	A floating point value (optional min./max. values)	Either slider or type-in field depending on the range	
InetAddress	An IPv4 or IPv6 address	Either four integer fields, each of value 0 - 255, or a type-in field with a lookup button.	 
Int	An integer value	Either slider or type-in field depending on the range	
Interview	A pseudo question used for sub-interviews; see `interview.callInterview(...)` in the API	For internal use only; never displayed	

TABLE 6-3 Question Types

Question Type	Description	GUI	Example
List	A list of complex values built from a set of questions.	A list box that displays the current contents of the list. The following questions add or edit a selected value in the list. This sequence is automatically terminated by a corresponding marker question.	
Null	Expository text; generally used to introduce a set of questions	Text only; no user input	
Properties	Enables configuring multiple key-value pairs in a single question.		
String	String information	Type-in field that optionally includes suggested answers	
StringList	A list of strings	A list box	
Tree	A tree selection	A tree selection GUI based on JTree	
YesNo	A convenience choice question for Yes/No answers	Radio buttons	

Designing Your Questions

Be sure to break down complex environment entries into simple values that can be answered by a single question, then build up the entry from those values. For example, if you are creating an environment entry that requires the name of a remote host and its port address, it's best not to ask for both pieces of information in a single question, but to ask for each piece of information in a separate question.

For example, the following entry previously seen in "Example 1" on page 51:

```
command.execute=com.sun.javatest.lib.ExecStdTestOtherJVMCmd
    C:\JDK\bin\java.exe -classpath $testSuiteRootDir\classes
    $testExecuteClass $testExecuteArgs
```

could be built up from a number of interview answers:

- Questions to determine whether the user plans to run the test locally or on a remote computer, and whether they plan to run the tests in the same JVM as the JavaTest Agent
- A question to determine the path of the Java runtime command
- One or more questions to determine the class path
- Questions that determine the path separator on the test platform

Landing Point Questions

You might find it convenient and useful to include questions that do not gather any information, but rather provide space between sections of the interview or provide a frame of reference to the user about where they are in the interview. You can use the Null question type for this type of interview question. In some cases you can use landing points as bridges between the main interview and sub-interviews.

Sub-Interviews

If your interview contains a large number of questions, you can break it up into sub-interviews. To create a sub interview, create a subtype of an `Interview` class. For example:

```
class MySubInterview extends Interview {
      ....
   }
```

The constructor should take a reference to the parent interview as an argument, and this reference should be passed to the superclass constructor. This identifies this interview as a sub-interview of the parent interview. For example:

```
MySubInterview(MyParentInterview parent) {
       super(parent, "myTag");
       ...
}
```

In the constructor, use the `setFirstQuestion` method to specify the first question in the sub-interview. Subsequent questions are found in the normal way using the `getNext` method. For example:

```
MySubInterview(Interview parent) {
```

```
    super(parent, "myTag");
    setFirstQuestion(qIntro);
}
```

By default, a sub-interview shares a resource file and More Info help files (see "Creating More Info" on page 70) with its parent interview (another reason to pass in that parent pointer). You can choose to use a different resource file and HelpSet if you want, although that is not typical for simple or moderately complex interviews. See the API specifications for `setResourceBundle` and `setHelpSet` for details.

At the end of the interview, have the last question return an instance of a `FinalQuestion`. This `FinalQuestion` is only a marker and does not have any question text, More Info, or a `getNext` method. For example:

```
Question qXXX = ......... {
   Question getNext() {
      return qEnd;
      }
   };
Question qEnd = new FinalQuestion(this);
```

For the parent interview to use a sub-interview, it must first create an instance of the sub-interview. This should be created once and stored in a field of the interview. For example:

```
Interview iMySubInterview = new SubInterview(this);
```

To call the sub-interview, use `callInterview` in a `getNext` method. The `callInterview` method takes two parameters — a reference to the interview to be called, and a follow-on question to be called when all the questions in the sub-interview have been asked. When the JavaTest harness sees the `FinalQuestion` at the end of a sub-interview, it goes back to where the interview was called and automatically uses the follow-on question that was specified there. For example:

```
Question getNext() {
   return callInterview(iMySubInterview, qFollowOnQuestion)
}
```

Flow Charts

Experience has shown that commercial flow charting tools can be very helpful if the interview becomes large and complicated. These tools can help you track the logical flow of the interview and keep track of sub-interviews.

Putting it All Together

To write a configuration interview, you must provide a class that implements the abstract class `InterviewParameters`. This class provides the JavaTest harness access to both the environment values and to the *standard values*. Standard values are configuration values used by the JavaTest harness to determine:

- Which tests in the test suite to run
- How to run them

To simplify this task, the JavaTest harness provides an implementation called `BasicInterviewParameters` that does a lot of the work for you. This class provides a standard prolog, questions for all the standard values, and a standard epilog. All you have to do is to implement the methods and questions for your test environment. However, you can also customize the other parts of the interview if you wish to.

The questions in `BasicInterviewParameters` are divided into the following groups:

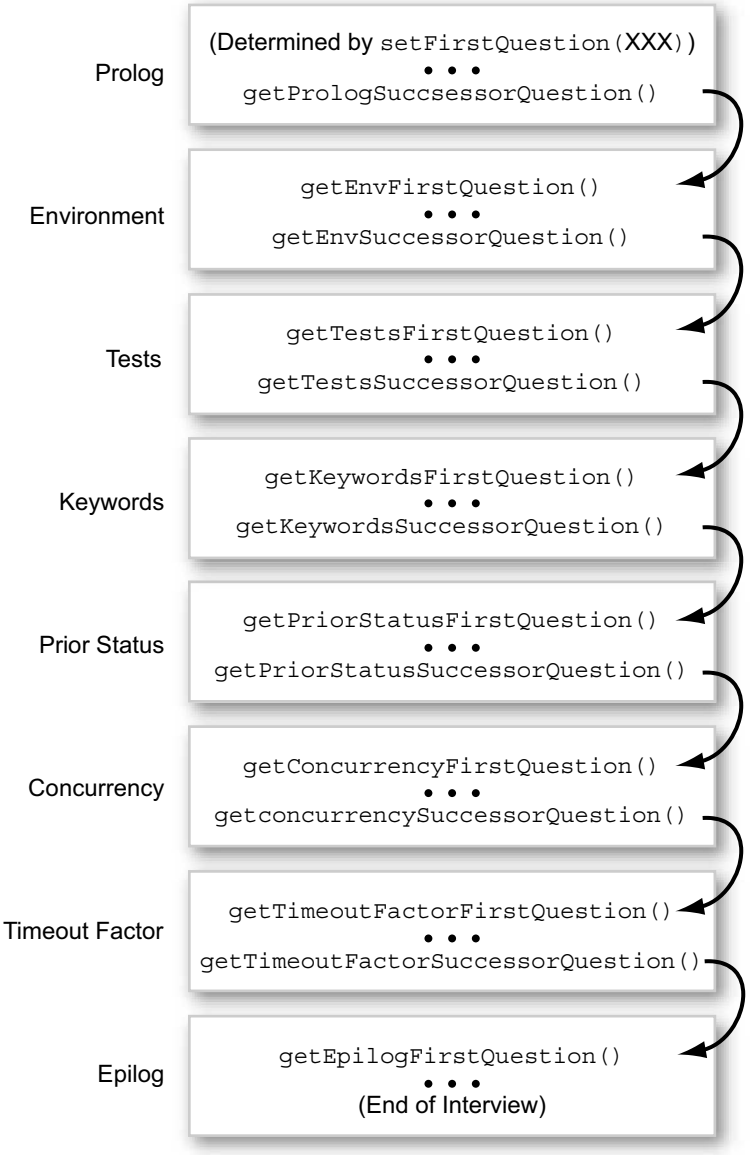
TABLE 6-4 Interview Question Groups

Group	Description
prolog	Identifies the interview and provides helpful information to the user about the interview such as how many questions the average interview consists of and how to proceed. Optionally, provides questions about the environment name and description.
environment	The questions you write to gather information for the test environment
tests	Allows users to specify sub-branches of test trees as a way of limiting which tests are executed during a test run
keywords	Allows uses to filter tests based on keyword values. Test suites can associate keywords with tests so that the keywords can be used as a basis for including and excluding tests from test runs.
prior status	Allows users to include and exclude tests based on their outcome in a prior test run. Test can be excluded and included based on the following status values: passed, failed, not run, error (test could not be run).
concurrency	Allows users to run tests concurrently on multi-processor computers
timeout factor	A value that is multiplied against a test's default timeout if a larger timeout is needed. The default timeout is 10 minutes.
epilog	Informs the user that they have completed the interview. May also provide information about how to run tests.

The groups of questions are presented in the order shown. Each group provides a method that identifies the first question in its group. The last question in the group uses another method to determine the next question. By default, the next question is the first question of the following group.

FIGURE 6-1 shows the “first” and “next” questions for each group of questions.

FIGURE 6-1 Interview Question Group First/Next Question Methods

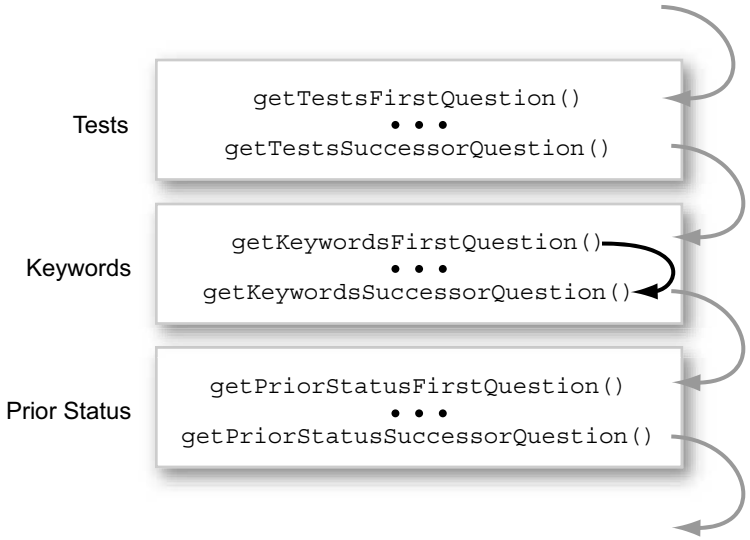


In most cases you only need to concern yourself with the environment group. For all the other groups, `BasicInterviewParameters` provides standard questions. If you find that you must customize the standard questions, you can replace the

questions for a group by redefining `get`*`Xxx`*`FirstQuestion()` to get your custom questions. In this case, you must also override the methods that provide access to these configuration values. See the API for more details.

If you find that any of the standard questions do not apply to your test suite, you can override the `get`*`Xxx`*`FirstQuestion()` question of any group you wish to skip so that it directly returns that group's `get`*`Xxx`*`SuccessorQuestion()`. This circumvents the code that executes the group's questions and jumps directly to the next group. For example, if your test suite does not use keywords, you can override the `getKeywordsFirstQuestion()` method and implement it so that it returns `getKeywordsSuccessorQuestion()` as shown in the following diagram.

FIGURE 6-2 Skipping the Keywords Standard Question



Providing the Prolog

The standard prolog always contains a standard welcome question; it also contains optional environment name and description questions. By default, the name and description questions are *not* displayed. You can enable the name and description questions by calling the `setNameAndDescriptionInPrologEnabled` method in your interview.

If the standard prolog questions do not meet your needs, you can override the prolog with one of your own. Specify your prolog by means of the standard `setFirstQuestion()` method of the interview. At the end of your prolog you must call the `getPrologSuccessorQuestion()` method to determine the first question of the next group.

Providing the Environment Group

This section describes the basic tasks necessary to write the environment portion of the interview. Unless your test suite requires you to make changes to the standard questions (prolog, standard values, epilog), the steps in this section describe what is required for you to produce your interview.

Put the group of questions that gather information for your test environment in your interview class. Remember to implement the `getEnvFirstQuestion` method to identify the first question of the group.

You must link the last question in the environment group to the rest of the interview (the standard values and epilog). In the `getNext()` method of the last question of your environment group, use `getEnvSuccessorQuestion()` to determine the next question in the interview — `BasicInterviewParameters` provides the rest of the interview.

Finally, you must implement the `getEnv()` method. The `getEnv()` method returns a `TestEnvironment` created from the responses to the questions. The easiest way is to call the interview's `export` method. The interview's export method calls the export methods for the questions on the current interview path. These questions export their values into a `Map` object from which you can construct a test environment. For detailed examples see the source code examples in the *jt_install*`\examples` directory.

Providing the Resource File for the Interview

In the constructor for your interview class, call:

`setResourceBundle(`*bundle_name***`);`**

For example:

```
setResourceBundle("i18n");
```

This uses a file called `i18n.properties` (or a localized variant) in the same directory as the interview class. See "Creating Question Text and More Info" on page 67 below for more information.

Providing the More Info Help for the Interview

In the constructor for your interview class, call:

`setHelpSet(`*moreInfo_helpset_name***`);`**

For example:

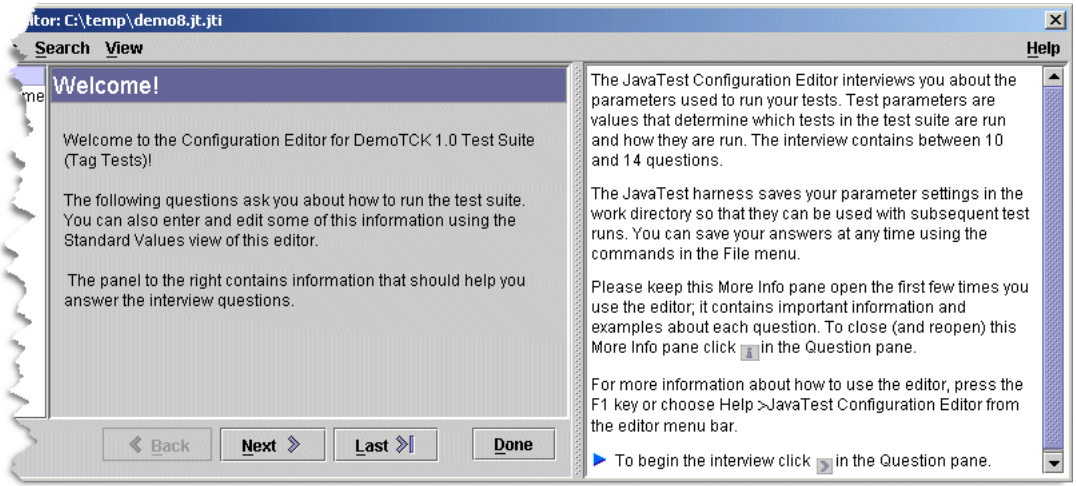
```
setHelpSet("moreInfo\demotck.hs");
```

This uses a HelpSet called `demotck.hs` (or a localized variant) in the `moreInfo` directory located in the directory that contains the interview class. See "Creating Question Text and More Info" on page 67 for more information.

Creating Question Text and More Info

As you saw when you ran the tutorial in Chapter 2, the configuration interview is presented to the user in the configuration editor. The question text and answer controls are presented in the Question pane, and information that helps the user answer the question is presented in the More Info pane.

FIGURE 6-3 The JavaTest Configuration Editor: Question and More Info Panes



The following sections focus on the text portions of the interview — the question text and the More Info help.

Writing Style

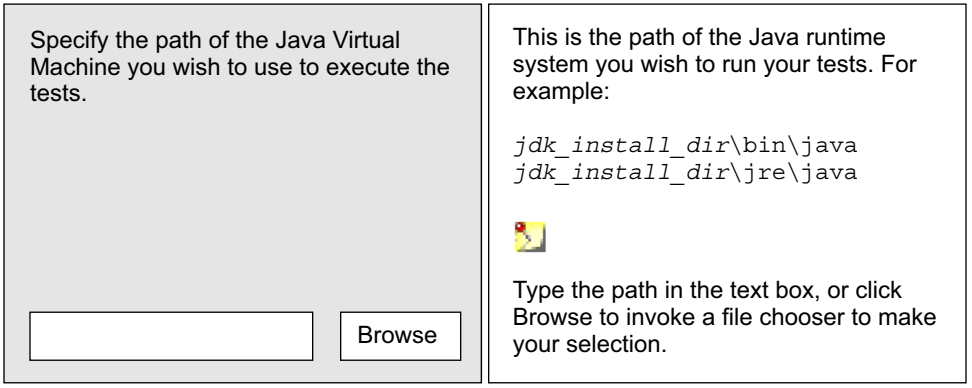
The style that you use for writing question text is very important. Experience has shown that it is very important to make the question text as clear, concise, and unambiguous as you can. Always try to use imperative statements, direct questions, and short explanations. If possible, have a proficient writer edit the questions to ensure readability and consistency.

Only put question text in the question pane. Information that helps the user answer the questions, including examples, should be provided in the More Info pane. The following figure shows a question where examples and other helpful information are included in the question pane with the question text:

FIGURE 6-4 Question without More Info Help

The following example shows how this question can be improved by reorganizing and slightly rewriting the question text and moving the examples and extra information to the More Info pane:

FIGURE 6-5 Question with More Info Help



There are a number of advantages to using the More Info pane to provide examples and other explanatory information:

- It allows you to keep the questions simpler. As users become familiar with the interview, they may no longer need the additional information to answer the questions. Displaying the extra information to the More Info pane moves it out of the way.
- The HTML-based More Info pane offers richer formatting, including: images, fonts, and tables
- The More Info pane can be scrolled to enable longer discussions and examples

Creating Question Text and Keys

Every interview question has its own unique key. The key is based on a name assigned by you and should uniquely identify the question with the interview. Normally, keys are of the form:

interview_class_name`.`*question_name*

You specify the *question_name* when you create the question, the *interview_class_name* is automatically added for you.

Question keys are used to identify question titles and text in resource files. The title of the interview and the title and text for every question in the interview is located in a Java resource file. The file contains the following types of elements:

- The title of the full interview
- A title for each question of the form: *question_key*`.smry`
- The text for each question of the form: *question_key*`.text`
- Additional entries for choice items that must be localized

For every interview question you create you must add corresponding `.smry` and `.text` entries into the resource file.

The following example shows a fragment of the Demo TCK configuration interview resource file:

```
title=Demo Interview Configuration Editor
AgentInterview.mapArgs.smry=Agent Map File
AgentInterview.mapArgs.text=Will you use a map file when you run the JavaTest
Agent?
DemoInterview.name.smry=Configuration Name
DemoInterview.name.text=Please provide a short identifier to name the
configuration you are creating here.
```

You can find the full Demo TCK configuration interview resource file in:

jt_install`\examples\javatest\simpleTags\src\i18n.properties`

The JavaTest harness uses the standard rules for accessing resource files. You can provide alternate versions for other locales by creating additional files in the same directory as `i18n.properties` with names of the form: `i18n_`*locale*`.properties`. See the Java SE platform resource file specification for more details.

Creating More Info

The JavaTest configuration editor enables architects and technical writers to present supplemental information for every question in the interview in the More Info pane. This information may include background information about the question, and examples and suggestions about how to answer them.

The More Info pane is implemented using an embedded JavaHelp window. The JavaHelp viewer supports HTML 3.2 with some additional extensions. For information about the JavaHelp technology, see:
`http://java.sun.com/products/javahelp`

Note – The JavaHelp libraries required to display More Info help are included in `javatest.jar` and should not be included separately.

The following procedures describe how to set up the More Info system for your interview and how to add More Info topics as you add questions to the interview.

▼ Set Up the More Info System

Create the directories and files used by the More Info system:

1. **Create a top-level directory called** `moreInfo`

   The `moreInfo` directory should be located in the same directory as your interview class file(s).

2. **Create directories named** `default` **and** `images` **in the** `moreInfo` **directory**

   The `default` directory contains the default localization. If your test suite is ever localized, the other locales can be added beside the `default` directory. The `images` directory contains any images you may use in the More Info help.

3. **Copy the Demo TCK HelpSet file to your `moreInfo` directory and rename it appropriately (retaining the `.hs` extension)**

   The HelpSet file is the XML file that the JavaHelp libraries look for to find all of the help information that defines the HelpSet. Rename it to reflect the name of your test suite. When you write your interview you specify the path to your HelpSet file.

   The path to the Demo TCK HelpSet file is:

   jt_install`\examples\javatest\simpleTags\src\moreInfo\demotck.hs`

4. **Edit the HelpSet file**

   The Demo TCK HelpSet file looks like:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE helpset

         "http://java.sun.com/products/javahelp/helpset_1_0.dtd">
<helpset version="1.0">
<!-- title -->
  <title>DemoTCK Configuration Interview - Help</title>
<!-- maps -->
  <maps>
     <mapref location="default/map.xml"/>
  </maps>
</helpset>
```

   Edit the contents of the `<title>` tag to reflect the name of your test suite.

5. **Copy the Demo TCK map file to the `default` directory**

   The JavaHelp map file is an XML file that contains a `<mapID>` entry for every More Info topic. The JavaHelp system uses it to assign an ID to every HTML file.

6. **Copy the Demo TCK style sheet to the `default` directory**

   Use the CSS, level 1 style sheet from the Demo TCK example for your More Info topics. Feel free to change it to suite your needs.

   The path to the Demo TCK style sheet file is:

jt_install`\examples\javatest\simpleTags\src\moreInfo\default\moreInfo.css`

▼ Create HTML Topics for All Interview Questions

For every question in your interview, you should create an HTML topic file and add an entry for that topic in the map file. The following steps describe how to do both:

1. **Create a map entry for the More Info topic**

   Every More Info topic file must have a corresponding `<mapID>` entry in the `map.xml` file. The JavaHelp system uses the IDs created in these files. The `target` attribute defines the ID, and the `url` attribute defines the path to HTML topic file (relative to the map file). The following example shows the map file for the Demo TCK test suite that you copied to your interview in a previous step.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map
  PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN"
         "http://java.sun.com/products/javahelp/map_1_0.dtd">
<map version="1.0">
<!-- More Info IDs -->
  <mapID target="DemoTCKParameters.cmd Type"            url="cmdType.html" />
  <mapID target="DemoTCKParameters.testVerboseLevel"    url=
"testVerboseLevel.html" />
  <mapID target="DemoTCKParameters.desc"                url="desc.html" />
  <mapID target="DemoTCKParameters.envEnd"              url="envEnd.html" />
  <mapID target="DemoTCKParameters.epilog"              url="epilog.html" />
  <mapID target="DemoTCKParameters.jvm"                 url="jvm.html" />
  <mapID target="DemoTCKParameters.name"                url="name.html" />
  <mapID target="DemoTCKParameters.prolog"              url="prolog.html" />
</map>
```

   Replace the `target` and `url` attributes to match your More Info topics. Remove any extra entries and add new entries as required.

2. **Create an HTML More Info topic file in the `default` directory**

   Copy one of the Demo TCK More Info files from:

   jt_install`\examples\javatest\simpleTags\src\moreInfo\default`

   and use it as a template. Be sure to include the `<link>` tag that references the style sheet.

   Experience has shown that it is helpful for the architect to create "stub" files for every question in the interview. These files are later completed by a technical writer and can provide information that the writer can use.

▼ Customizing Standard Question More Info

The JavaTest package includes default versions of the More Info HTML topics that describe the standard interview questions in both Question mode and Quick Set mode. However, should you wish to customize the content for some or all of these questions, you can override the defaults with files of your own. The following steps describe how to substitute your More Info topics for one of the standard interview questions:

1. **Determine the More Info ID for the question**

   You will override the More Info ID in your interview HelpSet. To do so, you have to determine the ID name of the standard question.

   a. **Open the configuration editor window to the question you wish to override**

      Make sure that you establish cursor focus in the question pane.

   b. **Press Alt-Shift-D**

      This opens the Configuration Editor Details Browser. The More Info ID is listed in the "id" field.

2. **Create a map entry in your map file as described in the previous section with the same name as the More Info ID you found in step 1.**

   For example:

```
<mapID target="TestsInterview.needTests"   url="my_needTests.html" />
```

   Note that the URL must contain the path to a file you create and must be included in your interview HelpSet.

3. **Create your custom version of the HTML More Info topic**

   Be sure that you create it at the location you specified in the map file URL field.

Creating the JAR File

After you have created your interview, you must package it into a JAR file for inclusion with your test suite. If you include other custom components with your test suite, they can be packaged together with the interview. See "The Test Suite JAR File" on page 33 for more information. You can also use the following file as an example:

jt_install`\examples\javatest\simpleTags\demotck\lib\jtdemotck.jar`

After you create the JAR file, put it in the test suite's `lib` directory and add it to the `classpath` entry in the `testsuite.jtt` file.

PART II Advanced Topics

The chapters in this part of the JavaTest Architect's Guide describe advanced features of the JavaTest harness that allow you to customize your test suite.

CHAPTER 7

Compiling Test Suites

Depending on how you design your test suite, you may be able to use the JavaTest harness to compile your tests when building your test suite. The instructions below describe how to compile tests if your test suite uses the Simple Interview and Standard Test Script (`StdTestScript`) as described in Chapter 12. To use the JavaTest harness to compile your tests you must:

- Specify the source files required to compile them in the tests' test descriptions
- Set two system properties on the JavaTest command line

System Properties

The following table describes the system properties used in compilation:

TABLE 7-1 System Properties Used in Compilation

System Property	Description	
`SimpleInterviewParameters.mode`	Specifies the operating mode used by this configuration. The valid values are:	
	certify	Executes test suite classes (assumes that tests are already compiled)
	precompile	Compiles the tests
	`developer`	Compiles the tests and runs them
`command.compile.java`	Specifies the command that the JavaTest harness uses to compile the tests.	

These properties are set on the JavaTest command line using the `-D` option, usually when running the JavaTest harness in batch mode in a shell script or batch command (see the JavaTest online help for details about batch mode).

You can use the following command lines as templates for compiling the simple test suite you created in Chapter 4:

Windows:

```
java  -DSimpleInterviewParameters.mode=precompile
      -Dcommand.compile.java='com.sun.javatest.lib.ProcessCommand
javac -d $testSuiteRootDir\classes -classpath $testSuiteRootDir\classes
       $testSource -jar javatest.jar -batch -testsuite ts_dir
      -workdir -overwrite work_dir -report report_dir
```

UNIX:

```
java  -DSimpleInterviewParameters.mode=precompile
      -Dcommand.compile.java='com.sun.javatest.lib.ProcessCommand
javac -d $testSuiteRootDir/classes -classpath $testSuiteRootDir/classes
       $testSource -jar javatest.jar -batch -testsuite ts_dir
      -workdir -overwrite work_dir -report report_dir
```

Note – Line breaks are added here to improve readability; your command line should be a single line that contains no line breaks.

The following table describes the parts of the command:

TABLE 7-2 Compilation Command Components

Component	Description
`com.sun.javatest.lib.ProcessCommand`	The library command used to run processes (in this case the Java compiler) on the same machine as the one the JavaTest harness is running. See for more information about this and other library commands.
`$testSuiteRootDir`	The JavaTest variable that represents the root directory of the test suite. This value is provided by the JavaTest harness when the command is executed. This value is taken from the path you specify to the `-testsuite` option.
`$testSource`	The JavaTest variable that represents the test source file to compile. This value is provided by the JavaTest harness.
`-batch`	Specifies that the JavaTest harness be executed in "batch" mode. When run in batch mode the JavaTest GUI is not started. See the JavaTest online help for more information.

TABLE 7-2 Compilation Command Components *(Continued)*

`-testsuite`	The fully qualified path name of the top-level test suite directory.
`-workdir -overwrite` *work_dir*	Specifies the name of the work directory to use for the compilation. The `-overwrite` option causes the JavaTest harness to first delete (if it exists) and create the specified work directory. It's generally a good idea to create fresh results each time you recompile.
`-report` *report_dir*	Specifies the name of the directory where reports are to be written. It is best to specify *report_dir* as a simple file name (no "`\`" or "`/`" characters; this causes the reports to be written in *work_dir*`\reports\`*report_dir*.

CHAPTER 8

The TestSuite Object

The JavaTest harness uses the `TestSuite` object as a portal to information about the test suite; whenever the JavaTest harness requires information about the test suite, it queries the `TestSuite` object. JavaTest reads the `testsuite.jtt` file to determine the class name and class path for the test suite; JavaTest then uses those properties to instantiate the `TestSuite` object. By default, the `TestSuite` object also gets a number of other standard properties from the `testsuite.jtt` file. As test suite architect, you create and maintain your `TestSuite` class and the `testsuite.jtt` file.

The `testsuite.jtt` File

The `testsuite.jtt` file is located in the top-level directory of the test suite and is a registry of information about the test suite that includes the paths to various JavaTest components as well as other static information about the test suite. The `testsuite.jtt` file generally contains at least two entries that tell the JavaTest harness how to start the `TestSuite` class:

- A `testsuite` entry that specifies the name of the `TestSuite` class and any arguments the class requires
- A `classpath` entry that specifies the class path on which the main `TestSuite` class can be found (typically, a JAR file that contains test suite-specific components). This entry is required if the `TestSuite` class or any other classes the `TestSuite` refers to are not located within `javatest.jar`.

The `testsuite.jtt` file usually contains other entries that specify information about the test suite; the JavaTest harness reads the file and passes the information to the `TestSuite` class when it starts. The following table describes the standard properties used by the `TestSuite` and may be specified in the `testsuite.jtt` file:

TABLE 8-1 `testsuite.jtt` Properties

Property	Description
additionalDocs	An optional list of resource names that identify JavaHelp helpsets for documents to be added to the JavaTest Help menu. The content of the helpsets must be provided on the test suite classpath (see *classpath* above). Example: `additionalDocs=jck.hs releasenotes.hs`
classpath	Extends the class path beyond `javatest.jar`. The class path is used to locate JavaTest plug-in classes (script, finder, interview) in JAR files, Zip files, or directories. You must separate entries with white space; relative paths are relative to the test suite root directory. If not given, classes must be located on the main JavaTest class path (not recommended). Always use "`/`" as the file separator. Default: Nothing in addition to `javatest.jar` Example: `classpath=lib/jtdemotck.jar`
env.tsRoot	A specialized entry to allow a legacy (prior to JavaTest version 3.0) test suite to override the values of `$testSuiteRoot` and `$testSuiteRootDir` that get set in the environment used to run tests. Most test suites should not need to set this property.
finder	The name of the test finder class and arguments (if any). This property is used by the default implementation of `TestSuite.createTestFinder` to determine the test finder to be used to locate tests. This property should be of the form "*classname args*", where *classname* identifies the name of the test finder class itself; any arguments are passed to the test finder's `init` method. Example: `testsuite=com.sun.javatest.finder.TagTestFinder` The default implementation of `TestSuite.createTestFinder` uses the following logic to determine the test finder: • If a `testsuite.jtd` file is found in the test suite `tests/` directory, or in the location specified in the *testsuite.jtd* entry of the `testsuite.jtt` file, the test finder is assumed to be `com.sun.javatest.finder.BinaryTestFinder` (which reads the `testsuite.jtd` file) • If a *finder* entry is found in the `testsuite.jtt` file, it is used to determine the test finder • If neither of the preceding are found, the default is to use `com.sun.javatest.lib.HTMLTestFinder` See the description of the *testsuite.jtd* entry below.

TABLE 8-1 `testsuite.jtt` Properties

Property	Description
id	A unique identifier composed of letters, digits, underscore, minus, and hyphen used to identify a specific version of a test suite. The JavaTest harness uses this property to ensure that component versions are compatible. By convention, the name is composed of the following parts: *technologyName*TCK_*version.* Example: `id=DemoTCK_tags_1.0`
initial.jtx	The path to the exclude list shipped with the test suite. If the path is relative, it is evaluated relative to test suite root directory. Always use "`/`" as the file separator. The recommended location for this file is in the test suite `lib/` directory. Example: `initial.jtx=lib/my_testsuite.jtx`
interview	The name of the interview class and arguments (if any). The default implementation of `TestSuite.createInterview` uses this property to determine the interview to use to obtain configuration information required to run the tests. The property should be of the form "*classname args*", where *classname* identifies the name of the interview class itself; any arguments are passed to the interview's `init` method. Example: `interview=com.sun.demotck.DemoInterview`
keywords	The list of valid keywords for this test suite. If the entry is present and contains a list of keywords, the keywords are added to the configuration editor keywords combo box. If the entry is omitted, it is taken to mean "unspecified" — in which case the user can use the configuration editor to specify keywords, but the configuration editor keywords combo box is disabled. If the entry is present but empty, it is taken to mean "none" — in which case the configuration editor does not present the keyword questions and tabs to the user.
latest.jtx	Specifies the location (as a URL) where the latest exclude list can be obtained. The `http:` and `file:` protocols are supported; authentication access is not yet supported. Example: `latest.jtx=http://my_company.com/support/exclude`
logo	Specifies the location on the class path of an image to be used as the test suite logo. The path is evaluated relative to the test suite root directory. This logo is displayed in the JavaTest Quick Start wizard.
name	The name of the test suite. This property is a string of up to 80 characters. By convention the name is composed of the following parts: *technology_name* TCK version	Test Suite [(*additional text*)] Example: `name=DemoTCK 1.0 Test Suite (Tag Tests)`

TABLE 8-1 `testsuite.jtt` Properties

Property	Description
script	The name of the test script class and arguments (if any). This property is used by the default implementation of `TestSuite.createScript` to determine the script to run the tests. The value should be of the form "*classname args*", where *classname* identifies the name of the `Script` class itself; any arguments are passed to the Script's `init` method. If this property is not specified, the default implementation of `TestSuite.createScript` reverts to the behavior defined for the JavaTest harness, version 2. Relying on this behavior is not recommended. Example: `script=com.sun.javatest.lib.StdTestScript`
serviceReader	Enables service management for the test suite. See Chapter 13 for detailed information about the service management feature.
testCount	The number of tests in the test suite. This property gives the JavaTest GUI a hint as to how many tests are in the test suite. Example: `testCount=450`
tests	By default, the JavaTest harness looks for test source files and test descriptions in the `tests/` directory in the test suite root directory. If you locate your test sources and test descriptions in a different directory, you must specify it using this property. Example: `tests=apitests`

TABLE 8-1 `testsuite.jtt` Properties

Property	Description
testsuite	Optional class name for a custom `TestSuite` class. The value should be of the form "*classname args*", where *classname* identifies the name of the `TestSuite` class itself; any arguments are passed to the `TestSuite` `init` method. The `TestSuite` class is used to access virtually all information about the test suite. Defaults to `com.sun.javatest.TestSuite`, which provides default behavior in concert with the `testsuite.jtt` file. Default: `testsuite=com.sun.javatest.TestSuite`
testsuite.jtd	Can be used to override the default location of the BinaryTestFinder data file. By default the TestSuite class looks for a file named `testsuite.jtd` in the directory specified by the "tests" property in `testsuite.jtt`. To override the default, specify the name and location of the BinaryTestFinder data file relative to the top-level directory of the product (location of the `testsuite.jtt` file). Example: `testsuite.jtd=tests/testsuite.jtd`
tmcontext	Optional class name for a custom `ContextManager` class. The value should be of the form *"classname"*, where *class name* identifies the name of the `ContextManager` class itself. The Test Manager (ExecTool) will query the test suite for this value as it builds the GUI. Defaults to `com.sun.javatest.exec.ContextManager`, which provides the default behavior of the harness. See Chapter 13 for more information on customization. Default: `tmcontext=com.sun.javatest.exec.ContextManager`

Note – The `testsuite.jtt` file is a Java property file and follows all the standard rules for Java property files defined in `java.util.Properties`.

The following example shows the `testsuite.jtt` file that is included with the tag example test suite.

```
# Test Suite properties file for DemoTCK test suite with
# tag-style tests

name=DemoTCK 1.0 Test Suite (Tag Tests)
id=DemoTCK_tags_1.0
classpath=lib/jtdemotck.jar
finder=com.sun.javatest.finder.TagTestFinder
script=com.sun.javatest.lib.StdTestScript
interview=com.sun.demotck.DemoTCKParameters
```

Overriding Default Methods

Although by default these properties are obtained from the `testsuite.jtt` file, you can override this behavior in your `TestSuite` class. By overriding the methods that get these properties, you can specify your own properties directly in the `TestSuite` class and/or manipulate the properties from `testsuite.jtt` as you wish. This is generally not necessary, but it is an option. Some reasons why you might choose to do this:

- To hide or protect some of the properties
- To determine some of these properties programmatically at runtime

To customize the TestSuite class, you must extend the base `com.sun.javatest.TestSuite` class. For details about which methods you may choose to override, see the `TestSuite` API documentation.

CHAPTER 9

Test Finder

After the `TestSuite` object is created, the JavaTest harness starts the test finder for the test suite. The `TestFinder` class creates the object responsible for finding and reading test descriptions — information required to execute a test is provided in its test description. At a basic level, the `TestFinder` class does the following:

1. Given an object (for example, a file) that contains a test description, the test finder uses the `read()` method to read the object. The `read()` method in turn calls the `scan()` method that must be provided by the test finder. The `scan()` method scans the file object for a test description or any references to other files that must be read (for example, if the file object is a directory).
2. The test finder creates one `TestDescription` object per test from the information it reads from the test description.
3. The `getTests()` method returns any test description information that it finds, and the `getFiles()` method returns a list of any other files that it must read to locate other tests.

Test Finder Subtypes

Because test descriptions can be embodied in a wide variety of formats, you can extend the base `TestFinder` class, customizing the `read()` method to meet the test suite's needs. The JavaTest Architect's release provides library classes that you can use directly; however, these classes do not provide much error checking. You can conveniently subtype a library class to provide additional validity checking.

Tag Test Finder

The `TagTestFinder` extends the `TestFinder` class. This class is also provided so that you can further extend and customize it to your own needs.

The `TagTestFinder` looks for test description information in specially commented tags in Java programs and shell scripts. The `TagTestFinder` recursively scans test directories looking for files with the `.java` extension and extracts test description information embedded directly in the test source files using specialized tags. These tags are located with Java language comments and begin with the @ character. The following figure shows an example of a file that contains tag test description entries.

```
/* @test
 * @bug 4105080
 * @summary Activation retry during a remote method call
 * to an activatable object can cause infinite recursion in
 * some situations.
 * @author John Brown
 *
 * @bug 4164971
 * @summary Allow non-public activatable class and/or
 * constructor Main test class has a non-public
 * constructor to ensure functionality is in
 * place
 *
 * @library ../../../testlibrary
 * @build TestLibrary RMID
 * @build ActivateMe CheckActivateRef_Stub CheckActivateRef
 * @run main/othervm/policy=security.policy/timeout=240
 */

import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

  public class CheckActivateRef
  extends Activatable
  implements ActivateMe, Runnable
{

  private CheckActivateRef(ActivationID id, MarshalledObject obj)
  throws ActivationException, RemoteException
  {
  super(id, 0);
  }
[...]
```

This format has the advantage of being very convenient for the test developer.

Examples of tag test descriptions can be found in *jt_install*`\examples\javatest\simpleTags\demotck\tests`.

HTML Test Finder

An example of a test finder that reads HTML test descriptions is the `JCKTestFinder` — a subtype of the `HTMLTestFinder` class that provides additional error checking. The `JCKTestFinder` is described in some detail here to demonstrate how a test finder works. `HTMLTestFinder` is provided with the JavaTest harness so that you can further extend it and customize it to your own needs.

Test suites that use the `HTMLTestFinder` class use HTML-based test descriptions to provide the information required to execute their tests. Distributed throughout the directories that contain the tests are HTML *test description files* that contain one or more *test description tables*. Each HTML test description table contains information about a single test (for example, its name in the class path). Every test must be represented by its own unique test description table; however, test description files can contain multiple test description tables. Test description tables are always assigned the HTML class "TestDescription" using the `class` attribute:

`<TABLE BORDER="1" class="TestDescription">`

The following HTML source produces the test description table that follows:

```
<table border="1" class="TestDescription">
<tr>
  <td>title</td>
  <td>Checkbox Tests</td>
</tr>
<tr>
  <td>source</td>
  <td>CheckboxTest.java</td>
</tr>
<tr>
  <td>executeClass</td>
  <td>javasoft.sqe.tests.api.java.awt.Checkbox.CheckboxTests</td>
</tr>
<tr>
  <td>executeArgs</td>
  <td>-TestCaseID ALL</td>
</tr>
<tr>
  <td>keywords</td>
  <td>runtime positive</td>
</tr>
</table>
```

TABLE 9-1 Test Description Table

Title	Checkbox Tests
source	CheckboxTest.java
executeClass	`javasoft.sqe.tests.api.java.awt.Checkbox.CheckboxTest`
executeArgs	`-TestCaseID ALL`
keywords	runtime positive

The `JCKTestFinder` test finder locates the HTML test description files by recursively scanning directories to look for files with the `.html` suffix, ignoring any other types of files. It reads the test description table, ignoring all information in the file except the content of the basic table tags.

If you include multiple test description tables in a single test description file, each test description table must be preceded by an `<A NAME>` HTML tag that provides a unique identifier for each test description table.

Note – Test description files should also contain comments and text that describe the test.

The `HTMLTestFinder` class can also check the validity of test description values. For example, the `HTMLTestFinder` can be run with flags that cause error messages to be printed if any of the test description fields are invalid. When you extend `HTMLTestFinder`, you can add your own validity checks.

The benefit of this format is that it makes it easy and convenient for users to browse test descriptions using the JavaTest harness GUI or a web browser. The trade-offs are that more work is required of the test developers to create and maintain the HTML files, and parsing these separate files has an impact on performance.

Examples of HTML test descriptions can be found in *jt_install*`\examples\javatest\simpleHTML\demotck\tests.`

Binary Test Finder

BinaryTestFinder was created to improve the startup performance of large test suites. It is capable of reading test description information from a highly optimized format created from any type of native test description.

The optimized format (*filename*`.jtd`) is created using a companion program called BinaryTestWriter. BinaryTestWriter uses a native test finder such as HTMLTestFinder, or TagTestFinder to find and read native test descriptions (for example, HTML files or source tags) and then creates a single, optimized file that contains the test

description information for all the tests in the test suite. If one is available for the test suite, the test suite uses the BinaryTestFinder to read test descriptions from that optimized file. Use of the BinaryTestFinder is highly recommended for larger test suites — it greatly reduces the time required to populate the JavaTest harness test tree.

BinaryTestWriter

BinaryTestWriter is a standalone utility that creates compressed file that contains a a set of binary test descriptions. It uses a test finder that you specify to locate the test descriptions for your test suite, and writes a compact representation of those test descriptions to a file that can be read by BinaryTestFinder (described in the next section).

BinaryTestWriter is run from the command line as follows:

`java -cp javatest.jar com.sun.javatest.finder.BinaryTestWriter` *arguments test-suite* [*tests*]

The following table describes the parts of the command:

TABLE 9-2 BinaryTestWriter Command Components

Component	Description
`-cp javatest.jar`	Puts `javatest.jar` on the class path

arguments	`-finder` *finderClass* [*finderArgs*] `-end`
	Specifies the test finder to use to locate the test descriptions in the specified test suite.
	finderClass: The name of the plug-in class for the desired test finder. The class must be on the class path used to run BinaryTestWriter.
	finderArgs: Any optional arguments passed to the test finder's `init` method.
	`-o` *output-file*
	Specifies where the set of compressed test descriptions is written. The output file always contains the `.jtd` suffix and is typically named `testsuite.jtd`. The `testsuite.jtd` file is usually placed in the test suite `tests/` directory.
	`-end`
	Defines the end of the finder specification

test-suite	The path to the directory in the test suite that contains the test descriptions (typically, the `tests/` directory)
tests	An optional list of directories in which to search for test descriptions (typically, directories under `tests/`)

Note – The *finderClass*, *finderArgs*, *test-suite* arguments are specified here exactly as they are when you run the JavaTest harness without using BinaryTestWriter.

BinaryTestFinder

BinaryTestFinder is a standard JavaTest test finder that knows how to read test descriptions stored in the file written by BinaryTestWriter. The full name of the class is:

```
com.sun.javatest.finder.BinaryTestFinder
```

The `BinaryTestFinder` class is provided in the standard `javatest.jar` file. You can use it through the standard string interface, or directly via the API. For details about the API, see the Javadoc documentation.

There are two ways you can use BinaryTestFinder:

- If you use the standard `TestSuite` class, you can place `testsuite.jtd` in the test suite `tests\` directory. If the file is found there it is used, otherwise the uncompressed files in this directory are used.
- Specify the finder explicitly in the `testsuite.jtt` file:

  `finder=com.sun.javatest.finder.BinaryTestFinder -binary` *testsuite*`.jtd`

  This method requires that `testsuite.jtd` be present when the test suite is run. If it is not present, the tests are not run and an error condition exists. You can use the `testsuite.jtd` property in the `testsuite.jtt` file to specify the location of the `testsuite.jtd` file. You must remember to run BinaryTestWriter before running the test suite.
- Override the `createTestFinder` method for the `TestSuite` class you provide for your test suite. This method allows you to dynamically determine whether to use BinaryTestFinder. The TestSuite class can check for the existence of the binary test description file (*testsuite*`.jtd`) before running tests; if the `.jtd` file is not found, it can choose to use an alternate finder.

Examples

The following example shows the command line used to start the basic non-customized TestFinder class:

```
java -cp lib/javatest.jar com.sun.javatest.finder.BinaryTestWriter
-finder com.sun.javatest.lib.HTMLTestFinder -dirWalk -end top_level_testsuite_dir/tests
```

This example shows the command line for starting a customized TestFinder class (MyTestFinder). The finder class takes `-dirWalk` and `-specialMode` as arguments. Note that the JAR file that contains the custom finder class (in this case `lib/mytck.jar`) is added to the class path.

java -cp lib/javatest.jar:lib/mytck.jar com.sun.javatest.finder.BinaryTestWriter -finder com.sun.*mytck***.lib.MyTestFinder -dirWalk -specialMode 2 -end** *top_level_testsuite_dir***/tests**

CHAPTER 10

Test Scripts

The test script is responsible for running a test, recording all the details in a `TestResult` object, and returning the test's status (pass, fail, error) to the JavaTest harness. The test script must understand how to interpret the test description information returned to it by the test finder. The test script breaks down the execution of the test into a series of logical steps based on information from the test description and the test execution model. The test script can run the test itself or delegate all or part of that responsibility to *commands*. A fresh, new copy of the test script is created for each test. This design allows you to create test scripts for different test suites that use the same commands, much like shell and batch scripts are composed from different operating system commands.

Design Decisions

One of the most significant design decisions that you make is how the test script executes tests. The mechanism that you design can be very simple but inflexible, or it can be more complex and much more flexible.

Simple Test Scripts

Simple and less flexible test scripts construct test command lines directly from the test description and the test environment.

At the most simplistic level, scripts can execute tests using `Runtime.exec`. For example using the JDK:

```
Runtime r = Runtime.getRuntime();
String[] cmd = {"java", "MyTest"};
String[] env = {"-classpath", testsDir + "/classes"};
```

```
Process p = r.exec(cmd, env);
// read output from test using
// p.getInputStream() and p.getErrorStream()
// (best done with a separate thread for each stream)
int rc = p.waitFor();
Status s = (rc == 0 ? Status.passed("OK") :
  Status.failed("process exited with return code " + rc);
// s contains result status from executing command
```

In this case the test script is responsible for collecting the test's exit status.

The JavaTest harness provides a number of library commands that the script can use to execute system commands in different execution environments; these are described in Appendix A. One example is the library command named `com.sun.javatest.lib.ProcessCommand`. `ProcessCommand` executes a system command in a separate process on the same machine running the test script. For example:

```
String[] args = {"-classpath" + testsDir + "/classes", "java", "MyTest"};
PrintWriter out1 = ... // create error message stream
PrintWriter out2 = ... // create output message stream
Command cmd = new ProcessCommand();
Status s = cmd.run(args, out1, out2);
// output from command will be written automatically to
// the out1 and out2 streams
// s contains result status from executing command
```

The result of the command is a `Status` object based upon the exit code of the process. The exit code is analyzed by the test script and factored into the final test result. For example, if a script is executing a test by means of a series of commands and one of them fails unexpectedly, the execution may stop at that point.

More Flexible Test Scripts

More sophisticated and flexible test scripts use *command templates* to create custom commands. Command templates are designed by you and are created by the configuration interview from configuration information and test description information (see Chapter 6). Command templates can be created with some components of the template specified in the form of variables that the test script resolves when it uses the command to run a test. A configuration interview may provide several different templates; the script chooses among them as required for each individual test.

For example, a configuration interview might create a custom command template named `command.testExecute` that can be used to run all of the tests in a test suite.

```
command.testExecute=com.sun.javatest.lib.ProcessCommand
\bin\java.exe -classpath $testSuiteRootDir\classesJDKC:\
$testExecuteClass $testExecuteArgs
```

The test script sets the value of the variables ($testExecuteClass and $testExecuteArgs) for *each test*. To review the parts of the template see "Example 1" on page 51.

The use of variables allows you to create flexible commands that can be used with all of the tests in the test suite. The following test script fragment shows how a test script invokes the `testExecute` command[1] whenever it runs a test. Note that the test script uses its `invokeCommand()` method to execute commands:

```
import com.sun.javatest.*;

class MyScript extends Script {
    public Status run(String[] args, TestDescription td,
TestEnvironment env) {

  ...

  // Extract values from the test description
  String executeClass = td.getParameter("executeClass");
  String executeArgs  = td.getParameter("executeArgs");

  ...

  // Set variables in the template
  env.put("testExecuteClass", executeClass);
  env.put("testExecuteArgs", executeArgs);
  // Invoke the command
  Status s = invokeCommand("testExecute");

  ...

  return s;
    }
}
```

In this example, the test script executes a single command for each test — the test scripts can also execute complex, multi-part tests that may involve multiple command invocations. The following examples describes some common multi-part test scenarios.

1. When the command is invoked, the "`command.`" prefix is not used.

Example 1

Compiler tests generally require a multi-part test script. To test the Java compiler two stages are required:

1. The compiler compiles test files
2. The output from that compilation is run to ensure that it executes as expected

Example 2

Distributed tests are required to start a process on a remote system with which the test interacts. This requires a multi-part test that:

1. Sets up the remote system
2. Runs the primary test class that interacts with the remote system

The JavaTest harness is shipped with the source to a sample test script (StdTestScript.java) that you can refer to in the *jt_install*`\examples\javatest\sampleFiles` directory.

See the `Script` API documentation for information about the `Script` class.

Writing Custom Commands

Commands are the means by which the JavaTest harness invokes platform or test components to perform a step of the test execution model embodied in a test script. The JavaTest harness provides standard commands that are suitable for most uses, including test systems that can execute programs in a separate address space, and test systems that provide a single Java virtual machine.

If none of the standard commands are suitable, you can write a new one tailored to the test suite's specific requirements. One scenario that requires a custom command is when the test suite uses a single JVM, and the test invokes a program that does not have a standard interface that can be used by one of the standard commands. In this case, you can write a very simple converter command that connects the interface expected by the JavaTest harness with the interface provided by the program.

The class for a command is similar (apart from the name) to the standard `Test` interface. The full class name is `com.sun.javatest.Command`.

```
abstract class Command {
   Status run(String[] args, PrintWriter out1, PrintWriter out2)
   ...
}
```

The `args` argument is constructed in and passed down from the script that invokes the command. Output written to the `out1` stream and `out2` stream is recorded in the appropriate test result file.

EXAMPLE 10-1 is an example of a command that invokes a compiler in the same JVM as the JavaTest harness, using an API for the compiler. The example uses the JDK compiler which is usually invoked directly from the command line; however, in this case an undocumented API is used. The details of how to create the `PrintStream outStream` from the `PrintWriter out` are omitted here for simplicity; the main point is to illustrate how easy it can be to write a wrapper class that passes arguments through to a non-standard API, and converts the results into the format that the JavaTest harness requires.

See the source code for `JavaCompileCommand` in the *jt_install*`\examples\javatest\sampleFiles` directory for a complete, commented example.

EXAMPLE 10-1 `JavaCompileCommand`

```
public class JavaCompileCommand implements Command
{
   public Status run (String[] args, PrintWriter out1,PrintWriter out2)
   {
      PrintStream outStream = ... // create stream from out
      sun.tools.javac.Main compiler = ;
            new sun.tools.javac.Main(outStream, "javac")
      boolean ok = compiler.compile(args);
      return (ok ? Status.passed("Compilation OK") :
            Status.failed("Compilation failed"));
   }
}
```

For information on the standard commands provided with JavaTest see Appendix A.

Test Result

To store test results, the JavaTest harness creates and maintains a `TestResult` object for each test. The test script stores information in a `TestResult` object while it executes a test. This information is presented to users in the JavaTest GUI and is useful when troubleshooting test runs. The more information the test script provides, the easier it is for the user to understand what occurred when the test was run.

The `TestResult` object contains the following elements:

Test description	The test description used for the test.
Configuration	The portions of the environment used to run the test. This information is displayed to the user in the Configuration tab of the JavaTest GUI.
Test run details	Information about the test run. For example, start time, end time, This information is displayed to the user in the Test Run Details tab of the JavaTest GUI. **Note:** The test script has access to this field and can write additional information using the TestResult API.
Test run messages	Test output messages. This section is written by the Script class's `invokeCommand()` method. This section contains at least two subsections, one for messages from the test script and one for each part of the test (if it is a multi-part test). This information is displayed to the user in the Test Run Message tab of the JavaTest GUI.

When a test completes execution, the JavaTest harness writes the results to the file *testname*`.jtr` in the work directory. Test result files are created in directory hierarchies analogous to the hierarchies in which the tests are organized.

See the API documentation for the `TestResult` class.

CHAPTER 11

Service Management

This chapter describes the `ServiceManager` (`com.sun.javatest.services`) component provided by the JavaTest harness and how test suite architects can use it to manage services that a test suite might require for test execution.

This chapter contains the following sections:

Description

A service is any unique component related to a test suite that is required for test execution and must be started in separate process or thread (such as RMI daemon or ORB service) before the test suite is run. Some TCKs might require many services. The `ServiceManager` component enables users to configure the services required to run tests.

Each test suite optionally specifies and describes a set of services that it requires to run tests. During the test suite load process, the `TestSuite` instantiates objects that implement the `Service` interface and uploads those objects to the `ServiceManager` component.

The `ServiceManager` component manages all aspects of the service life cycle. From information given by the test suite and the JavaTest harness, it determines which services are required and when they should be started. From test execution events, it also determines which services should be stopped.

The `ServiceManager` is available either when running the JavaTest harness or as a separate service without running the JavaTest harness.

TABLE 11-1 Service Manager Features

Features	Description
Automatically start and stop services	Can start and stop services automatically.
Mapping services on tests or test suites	Mapping services on individual tests or a set of tests, not on the whole test suite, enables the ServiceManager to start/stop services for group of tests. For example, if a user is not running CORBA tests, the Service Manager will not try to start CORBA services. Consequently the user should not have to configure CORBA services.
Manually start a service	In case of remote test execution, users need the ability to determine (manually or automatically) if a service should be started or not.
Services are thread safe	Services work safely in case of parallel test execution.
Process management - when to start services	Provides an ability to start services in two modes: 1. As needed - Start service only when the first test or group of tests that needs the service is about to run. 2. Up front - Start all needed services up front so that any errors starting services can be detected before the actual test run.
Process management - when to stop services	Needed services stay up, once started, until the end of the run or until it can be shown they are no longer be required. Test execution finish, time out, and end of test run are points to stop related services.
Performance	The test suite does not run noticeably slower with this mechanism enabled.
Usability - configuration file	The user only provides or edits a single configuration file for all the services, not a file for each service. The file is optional. If the user doesn't provide a file, the test suite should assume that any required services will be started manually.
Services dependencies	Dependencies between different services are supported. For example, one service must be started or stopped before other services.

Services-Related Work Flow

Services-related work flow of harness execution is supported in both GUI and batch mode test execution. The work flow consists of the following:

1. The `ServiceManager` and `Service` instances are instantiated after the test suite is loaded inside harness.

2. When the JavaTest harness object (`Harness`) starts a test run, information that the `Harness` has about the tests and groups of tests that should be run is provided to the `ServiceManager`.

3. The `ServiceManager` filters out services that are unnecessary for the test run based on information from the `Harness` and information from deployed services regarding which test paths for which the service should be started.

   The services-related workflow is performed before starting a test run in the main `Harness` execution thread. During this process one of the following actions is taken based on the information that the harness has about the tests and groups of tests that should be run:

   - Start services as needed.

     After being notified by the `Harness` that a test has started, the `ServiceManager` determines if a particular service should be started for running the next test. This enables "lazy" service start up, which may be useful for performance reasons.

   - Start all required services now.

     Before running any tests, the `ServiceManager` starts all required services.

   - Start services manually.

     Service management is turned off for the test run. The `Harness` assumes that the user will manually start all required services.

Note – When running in GUI mode, the `ServiceManager` functionality is enabled after the user presses the Run button which blocks the `Harness` execution thread until it determines how services will be started. In batch mode, the `ServiceManager` is functionality is enabled by using an option in the command line.

---

4. The `ServiceManager` stops services either as it determines that a service is not required (all tests that require this service are completed) or at the end of test run.

   Stopping services after the test run finished is preferred.

Implementation

Because the `ServiceManager` component must serve different requirements, its service support design is as flexible as possible.To achieve this flexibility, the design uses abstract concepts of service, service description, service execution and service parameters. Some functionality, such as remote service management when services are instantiated and run on a remote host, has not been implemented. However the capability to implement the functionality is supported in the architecture. Additional functionality could be implemented by the test suite and set through the API, as is currently done in the JavaTest harness for other components.

Note – Services support is optional, so that test suites, which do not need services support, are not required to implement additional components.

The JavaTest harness provides useful default implementations of the following interfaces:

- `Service` - Interface describing a service.

  `Service` objects should be instantiated by `ServiceReader`. The particular implementation class should be specified so that the `ServiceReader` can access it. A default implementation, `AntService` (see "Implementation of `Service` Interface" on page 106) is provided by the JavaTest harness. Such a service is the Ant target in provided ant `xml` file. The benefit of such a service representation is that the service can easily be started without running the JavaTest harness.

- `ServiceReader` - Interface responsible for reading service definitions and parameters.

  The implementation should be provided by the test suite. A default implementation, `XMLServiceReader` (see "Implementation of `ServiceReader` Interface" on page 105), is provided by the JavaTest harness. This implementation reads `Service` type definitions and start parameters of each particular instance, as well as maps test paths to service instances from a single XML document.

- `ServiceConnector` and `ServiceExecutor` - `ServiceConnector` is responsible for the connection between harness representative (`Service` interface) and `ServiceExecutor` is responsible for responsible for running service in case of remote execution.

  Each executor type is related to a respective service type, such as `AntService` and `AntServiceExecutor` (see "Implementation of `Service` Interface" on page 106). `Connector` doesn't depend on the `Service` and `ServiceExecutor` type. `Connector` has a unified interface and any `Connector` implementation should work with any `Service` - `ServiceExecutor` pair. The JavaTest harness

only provides a pseudo local connector that redirects requests to a `ServiceExecutor` working in the same VM. The following are available types of `ServiceExecutor`:

- `ThreadServiceExecutor` - Executes any service described by `Runnable` object in a separate thread.
- `ProcessServiceExecutor` - Executes in a separate process by `Runnable.exec()`.
- `AntServiceExecutor` - Extends `ProcessServiceExecutor` to execute `ant` tasks using Ant.

Implementation of `ServiceReader` Interface

To make the process of acquiring information about services and instantiating components more flexible, a test suite should provide a special component that implements the `ServiceReader` interface. The `ServiceReader` interface reads information regarding service descriptions and tests-to-services mappings during test suite load, instantiates `Service` objects, and pushes the collected information into them.

The JavaTest harness provides a default implementation (`XMLServicesReader`) of the `ServicesReader` interface. `XMLServicesReader` looks for information inside one XML file. The following sample provides a description of the contents and format of this file.

```
<Services>
<property file="local.properties"/>
<property name="port" value="8080"/>
<property name="testsuite" value="${root}/tests"/>
.................................
<service id="rmid_1" class="com.foo.bar.RMIDService" description="This is first
variant of service to start rmid daemon">
<arg name="arg1" value="5000"/>
<arg name="arg2" value="${testsuite}"/>
</service>
<service id="rmid_2" class="com.foo.bar.RMIDService" description="This is second
variant of service to start rmid daemon">
</service>
<service id="msg_service" class="com.foo.bar2.MessagingService" description=
"messaging service">
<arg name="timeout" value="1000"/>
</service>
.................................
<testpath path="api/java_rmi">
<service refid="ant"/>
<service refid="rmid_1">
```

```
</testpath>
<testpath path="api/foo_bar">
<service refid="rmid_2"/>
<service refid="msg_service"/>
</testpath>
.....................................
</Services>
```

The format of the `XMLServicesReader` file consists of three sections:

- **Properties** - The first section of the file specifies the properties.

  You can load property values from a file or specify them separately. Properties that do not have explicitly-set values are called parameters. Parameter values are resolved later by `Service` or `ParamResolver` classes. In the code example, `root` and `testsuite` properties are parameters. The `Service` interface should provide operations to get and set parameters and arguments. `Service` properties are described in "Service Properties" on page 107.

- **Services** - The first section of the file describes the services.

  Services are described by using a tag. Each service specification tag contains a unique string ID that enables the user to refer to this service in the test map section, a `Service` interface implementation class, and description text.

- **Test Map** - The third section of the file provides a map from test paths to services.

  Based on this information, `ServiceManager` determines, which services should be started/stopped and when. It consists of a regular expression with path-to-tests pattern and tags with references to services. One test path can refer to many services. Different test paths can refer to the same services. In case both such test paths are selected for test run, only one instance of a service will be started.

Implementation of `Service` Interface

The JavaTest harness provides a default implementation (`AntService`) of the `Service` interface that not only provides a description but also provides a definition and execution. The default implementation uses Ant. Each service is presented as an Ant target in a valid Ant file. In the service description XML file, a special service class (`com.sun.javatest.services.AntService`) describes the Ant-based service as follows:

```
<servicedef id="any_uniq_id _you_want"
class="com.sun.javatest.services.AntService">
<arg name="ant.bin" value="~/apache-ant/bin/ant"/>
<arg name="ant.targets" value="rmid-target run-tests"/>
<arg name="ant.workdir" value="directory_to_start_ant_from"/>
<arg name="ant.env.JAVA_HOME" value="path to JDK"/>
<arg name="option1" value="-buildfile ${lib}/build.xml"/>
```

```
<arg name="option2" value="-verbose"/>
</service>
```

- `ant.bin` - Specifies path to ANT execution script.
- `ant.targets` - Specifies targets to execute
- `ant.workdir` - Specifies the directory from which to start ant.

This is set to `java.lang.ProcessBuilder` using its `directory()` method.

- `ant.env.JAVA_HOME` - The environment entry with which the process starts.

  Set to `ProcessBuilder` through its `environment()` method.

- "`option1`" and "`option2`" - All other arguments inside the `AntService` are interpreted as ant start options.

  No special naming conventions are needed for them.

- Ant-based services are the only service implementations provided by JavaTest harness. JavaTest harness provides `AntService`, which implements the `Service` interface, and `AntServiceExecutor`, which implements the `ServiceExecutor` interface.

Service Properties

All possible parameters, commands, and arguments, could be described as `string` key-value pairs. Such pairs should be passed to the `Service` object by the `ServiceReader`, who creates the `Service` objects. Each particular `Service` type may interpret them and pass through a connector to the `ServiceExecutor`.

However, not all key-value pairs or values may be known when the test suite is created. For example, some host names and port numbers may be known only when answering an Interview question before a test run. To resolve this issue, values in service descriptions are parametrized. That means, that some key-value pairs will have variable values with references to other keys, and are resolved when additional non-variable sets of properties are passed from somewhere outside (such as after interview completion).

The `ServiceManager` has a public API used to access to each `Service` and its `ServiceProperties` object. The non-variable set of properties may be passed at any time. However, a more convenient approach is to resolve variable properties using values obtained from the interview. These values are also key-value pairs and are stored in a special object, `TestEnvironment`. `Harness` passes this object to resolve service properties before every test run. Consequently, refer to interview question's tags to resolve variable values by interview.

A special `ServiceProperties` class performs this behavior. Its instance is the main part of each `Service` object. `ServiceReader` should instantiate a `ServiceProperties` object, fill it with information available from the service description file and pass it to the corresponding `Service` object. Should the test suite use the standard `XMLServiceReader`, the test suite developer shouldn't care about this.

Each key-value pair from the `ServiceReader` must use the format:

`string=[string | ${string}]`*

If a key has no value, it becomes a variable.

`${string}` represents a reference to another key. If its value has at least one reference inside it, it also becomes variable.

Example:

```
key1=
key2=value2
key3=value31_${key1}_value32_${key2}_value33
```

Later if we pass `key1=value1`, the expression is resolved as:

```
key1=value1

key2=value2

key3=value31_value1_value32_value2_value33
```

As described in "Implementation of `ServiceReader` Interface" on page 105, some properties are common to several objects and some are individually specified for each Service object. That is the reason why there are two namespaces for property names. One namespace is a common namespace, and the other is an individual namespace. The common namespace is shared between all `Services`. Consequently, a property, specified inside each particular `Service`, may refer to common properties. If a name of a property specified for an individual Service is contained in a common namespace, the individual property overwrites the common property.

Individual namespaces are not shared between `Service` objects. A property from one individual namespace cannot refer to a property from another individual namespace. If a property attempts to do this, the reference is interpreted as unknown.

When it prepare a command, a `Service` objects asks its `ServiceProperties` object to resolve its arguments. The `ServiceProperties` object returns a Map containing only the resolved individual properties (not the common properties). Resolved common properties may be achieved, using another method. Such division

enables the `Service` class implementation to be simplified. It treats its individual arguments by using its own name conventions. Common properties are used to resolve individual properties or for other goals.

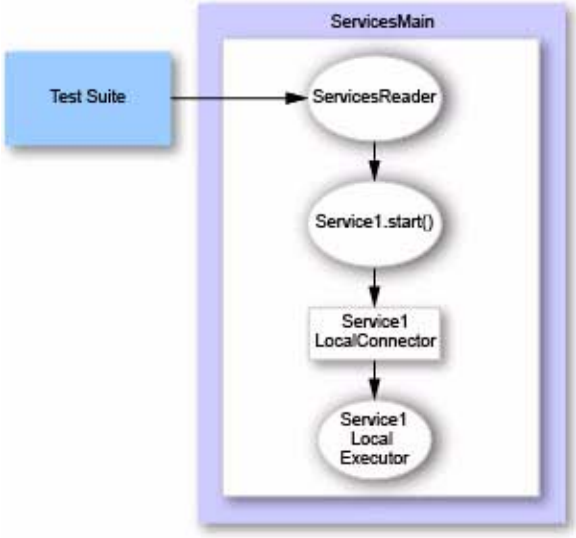
Service Management Architecture

The architecture of the Service Management feature of the JavaTest harness consists of five components:

- `ServiceManager`
- `Service`
- `ServiceExecutor`
- `Connector`
- `ParamResolver`

FIGURE 11-1 illustrates the relationship between these components.

FIGURE 11-1 Service Management Architecture



1. `ServiceManager` - Instantiated for each test suite instance.

   The same test suite, opened two times in different tabs, is interpreted as two different test suites and will have different `ServiceManager` objects. `ServiceManager` objects accomplish the following functions:

- Are notified of all `Harness` events.
- Manage a set of services.
- Provide methods to achieve service's state, info and data to the JavaTest harness and test suite.
- Provide methods for external configuration by JavaTest and test suite.
- Start and stop services
- Count which services and when should be started and stopped.

2. `Service` - Root service interface.

   `Service` interface has two main goals:

   - Contains service information, execution parameters, and test mapping.
   - Provides start, stop, other operating methods, are invoked by `ServiceManager`, and are delegated through `Connector` to `ServiceExecutor`

3. `ServiceExecutor` - Root interface for the service executor.

   Method invocations from `Service` go through `Connector` and are executed by `ServiceExecutor`. The `Service` and `ServiceExecutor` implementation types have a 1:n relationship. Consequently, each `Service` implementation can have different `ServiceExecutors` for different situations (such as local and remote execution). `ServiceExecutor` implementations can execute a service as separate process, as a thread, or in any other required manner. `Service` and `ServiceExecutor` types must be coordinated to perform message exchange correctly.

4. `Connector` - Interface that determines common connection methods between `Service` and `ServiceExecutor`, such as `sendMessage` or `getOutputStream` methods.

   Particular implementation should not be related with concrete Service and ServiceExecutor realizations. `Connector` is harness-side component, and we have no any interface for agent side part, because on those side such component (and it's incoming events) manages `ServiceExecutor`. Agent-side component is not under any management, so there is no need for it to have API.

5. `ParamResolver` - Component, related with `ServiceExecutor`. `Connector`, that sends commands to a service and provides the parameters for this command execution.

   The parameters are decoded by `ParamResolver` and passed to `ServiceExecutor`. For example, if a connector sends `"$host_name" param`, it should be resolved by the `ParamResolver`. Implementations of `ParamResolver` should be interoperable with `ServiceExecutor`. How and what to resolve depends on the implementations of both components.

Service execution is divided into 3 components (`Service`, `Connector`, `ServiceExecutor`), because it must be able to implement remote services start-up and execution by any test suite. It is not possible to implement this feature directly in the JavaTest harness and its agent, as requirements from different customers vary.

Mechanism to Instantiate `Service`, `Connector`, and `ServiceExecutor` Interfaces

The `Connector` and `ServiceExecutor` may differ because configuration settings (such as local or remote execution) and the specific implementors are known only at the beginning of a test run. The `ServiceManager` should have a public API to enable any object (such as a test suite) to specify non-default `Connectors` for each service.

The `Service` interface has a method that returns an instance of the default `ServiceExecutor`, which will be used should service be run in the same VM as the JavaTest harness. This executor interoperates with the pseudo LocalConnector, which directly delegates requests to this executor by invoking its methods. If a test suite wants to execute a service in another way, before starting test run, it should set another `Connector` for this `Service` (through the `ServiceManager` by using the Service's ID). This `Connector` may be directly associated with a `ServiceExecutor` (as `LocalConnector` does it), or it can be a network connector, and send messages to a remote agent.

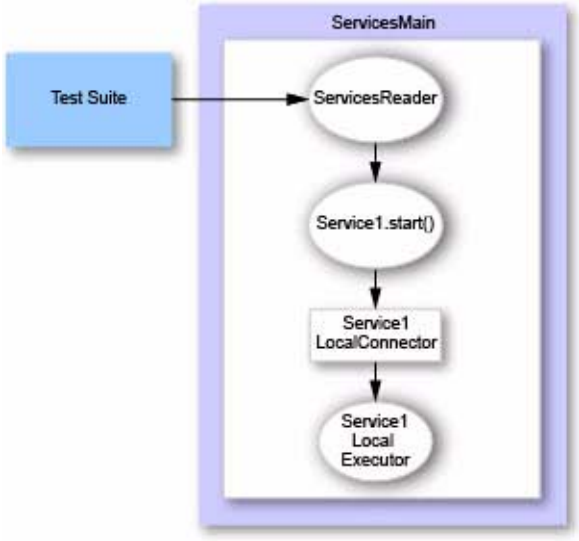
Separate Services Start Up

To simplify service start-up (in case there is no remote environment or you don't want to use the JavaTest harness to run the Service management feature), a separate entry point is available inside the JavaTest harness, such as `ServicesMain`, that performs the following operations:

1. Takes the test suite path as input.
2. Instantiates all found `Services`, `ServiceExecutors`, and local `Connectors`.
3. Invokes the `Service.start` methods.

Services are unmanageable in this case and must be stopped by shutdown hook. FIGURE 11-2 illustrates the sequence of performing a separate services start-up.

FIGURE 11-2 Separate Service Start-Up



CHAPTER 12

Running JUnit Tests

This chapter explains how to retrofit existing JUnit 3.x or 4.x test suites to enable them to run with JavaTest Harness. This information can also help you author new JUnit tests that run under the harness.

JUnit is a simple framework for writing and running automated tests. Written by Erich Gamma and Kent Beck in 1997, JUnit exposed test driven development practices from the Smalltalk world into the Java programming community. JUnit is now an open-source project at SourceForge.net.

The JUnit framework provides a fast, simple interface for creating a set of tests and executing them by a common method (for example, using Ant or a shell script). The framework places very few restrictions on what the tester must do to write a test class. The core JUnit distribution has few facilities for GUI interaction or reporting, and it has no robust interface for configuration.

The procedure described here enables JUnit tests to be run under the JavaTest Harness harness. The JavaTest Harness harness provides a robust GUI interface, many reporting options, and an opportunity to build a robust configuration system for the tests. The harness can be configured to allow customization of the GUI, report types, result presentation, and more. These services might be useful for users who want to wrap more powerful facilities around their existing test infrastructure.

The Retrofit Process

This section describes the process of retrofitting JUnit tests so that they run on the JavaTest Harness harness.

Prerequisites for Converting Tests

To undertake a conversion process, you must be familiar with some of the inner workings of the JUnit test suite you are converting. Specifically, you need to know:

- How the JUnit tests can be distinguished from other tests.
- The version of JUnit that works with the test suite (3.x or 4.x).
- Where the tests are stored. For example, are they in a single directory tree?
- The libraries or supporting processes required to run the tests.
- The configuration settings or files necessary to run the tests.

Tests written to work with JUnit 3.x are typically identified as being a subclass of `junit.framework.TestCase`. To find JUnit 3.x tests, use the `com.sun.javatest.junit.JUnitSuperTestFinder` class (located in the `jt-junit.jar` archive) to scan the test classes. Each class that is a subclass of `junit.framework.TestCase` is designated as a recognized test.

JUnit 4.x style tests do not use a specific superclass, rather, they tag classes with the `org.junit.Test` annotation. The JavaTest Harness harness library `jt-junit.jar` provides the class `com.sun.javatest.junit.JUnitAnnotationTestFinder` to find 4.x style tests. It operates much like the `JUnitSuperTestFinder` class, but looks for different criteria.

See "JUnitSuperTestFinder" on page 117 and "JUnitAnnotationTestFinder" on page 118 for more details.

▼ Procedure for Converting Tests

This procedure describes how to set up files, property settings, and configuration settings before running a JUnit test.

1. **Create a `testsuite.jtt` file in root of the product directory.**

   For example, if the product unpacks into directory `foo/`, the `testsuite.jtt` file should be in that directory. It does not necessarily need to be co-located with the tests.

   The `.jtt` file is a properties formatted file, with `key=value` pairs on each line. Setting the `name` and `id` keys is mandatory. The `name` is a short descriptive name for your test suite, the `id` is an internal key used identify this test suite.

2. **Select your method for scanning for tests by specifying a `TestFinder` class.**

   The line for specifying the `TestFinder` class looks like this:

   ```
   finder = com.sun.javatest.junit.JUnitSuperTestFinder
   ```

```
-superclass junit.framework.TestCase
```

See "JUnitAnnotationTestFinder" on page 118 and "JUnitAnnotationTestFinder" on page 118 for further information.

3. **Select your `TestSuite` class, using `com.sun.javatest.junit.JUnitTestSuite` if you do not subclass it.**

   Use a fully qualified class name. This class must be available on the system class path, preferably on the class path defined in your `.jtt` file. For example:

   ```
   testsuite = com.sun.javatest.junit.JUnitTestSuite
   ```

4. **Specify the interview.**

   If you don't have your own interview, use the line below as the default. This class must be available on the system class path, preferably on the class path setting in your `.jtt` file. For example:

   ```
   interview = com.sun.javatest.junit.JUnitBaseInterview
   ```

5. **Provide a `tests` setting.**

   The tests location is important because it is forwarded to the `TestFinder` class you specified in Step 2. This location is often relative to the location of the `testsuite.jtt` file as described in Step 2. Use forward slashes to make the path platform independent. Do not use absolute paths or relative paths to places above `testsuite.jtt`. One of the following lines might serve as an example:

   - If you are scanning `.java` files, they might be located below the `tests/` directory.

     ```
     tests = tests
     ```

   - If you are scanning class files, they might be located below the `classes/` directory:

     ```
     tests = classes
     ```

   See "JUnitSuperTestFinder" on page 117 and "JUnitAnnotationTestFinder" on page 118 for further information.

6. **Make sure that the paths to any classes you specify in the `testsuite.jtt` file are assigned to the `classpath` key in the `testsuite.jtt` file.**

   This how the harness locates the classes. For example, if you specify:

   ```
   interview = com.sun.javatest.junit.JUnitBaseInterview
   ```

   be sure to add the path to the JAR file that contains that class to the `classpath` key as shown here:

   ```
   classpath = lib/jt-junit.jar lib/jt-myts.jar
   ```

7. **Try running the harness to see if it finds your tests.**

   You have to decide how to arrange your (JAR) files and resolve paths. The general form is:

   ```
   > cd mytestsuite/
   > java -jar lib/javatest.jar -ts .
   ```

   This starts the harness and forces it to load the test suite located in the current directory (represented by "."). The `testsuite.jtt` file must be located in the "." directory.

   When the main window comes up, you should see a tree populated with the tests you intended. Check the counters on the main panel to view a summary of the tests that were found. You can check the View > Properties menu item to verify that the plug-in classes are loaded as you expect.

Technical Details

This section describes the two main sets of classes that provide JUnit support. The first is the `JUnitTestFinder` (a subclass of `com.sun.javatest.TestFinder`). Variations of the `JUnitTestFinder`, `JUnitSuperTestFinder` and `JUnitAnnotationTestFinder` classes roughly correspond to JUnit 3.x and 4.x support. The difference is explained below.

The second supporting component is the `JUnitMultiTest` class that is responsible for executing the tests.

Support Classes

The following additional "glue" classes are provided to connect everything: `JUnitTestSuite`, `JUnitBaseInterview`, and `JUnitTestRunner`. Each supporting class is explained below.

- The `JUnitTestSuite` class is a very simple class that instructs the harness to use the `JUnitTestRunner` to execute tests. If this method is not present, the `DefaultTestRunner` is used. This is the traditional way to execute tests requiring a `Script` class. Because the `TestRunner` class is present, there is full control over how the tests are executed. For example, the harness can determine how many tests are run simultaneously and how they are launched (for example, using `exec`). By extending this class, you have access to override other aspects of the harness. See the `TestRunner` API for more information. Note that many of

the settings that this document describes in the `testsuite.jtt` file can be hard coded into the `TestSuite` subclass. The `TestSuite` base class provides the functionality to instantiate the settings in the `testsuite.jtt`.

- The `JUnitBaseInterview` class is a skeleton interview class that does not require any input from the user. If your JUnit tests do not require a setting from the user, do not modify it. Try one of the following methods to get values from the user:
  - Read a configuration file from a pre-determined location, perhaps a location relative to the test suite root (`TestSuite.getRootDir()`).
  - Ask the user for the values directly using the `com.sun.interview` API. This is the primary means by which the harness is designed to get values from the user. In either case, the value(s) must end up in the `Map` provided in `Interview.export(Map)`. The `Map` is the set of values that the other classes must have access to, namely the `JUnitTestRunner` and classes it creates (`JUnitMultiTest`). Read Chapter 6 for more information.
- The `JUnitTestRunner` class is responsible for dispatching tests. It has access, via an `Iterator`, to the entire list of tests to be executed during a test run. Because a test is represented by a `TestDescription`, you must customize your test finder to add any settings that you will want later (in this class). The default implementation executes the test using `JUnitBareMultiTest` if the `TestDescription` property `junit.finderscantype` is set to `superclass`. If it is not set to `superclass`, it uses the `JUnitAnnotationMultiTest`class. You may want to change this behavior, use your own `JUnitMultiTest` class, or a subclass of one of these.

JUnitSuperTestFinder

This class looks for a superclass that identifies the class as a JUnit test. By default it searches the ancestors of each class for `junit.framework.TestCase`. Because a test suite might require further derivations of `junit.framework.TestCase` to support its particular needs, you can use the `-superclass` option to specify a more specific class.

For example, consider the following class structure:

```
java.lang.Object
   junit.framework.TestCase
      foo.BetterTestCase
         product.Test0002a
```

`Test0002a` is a subclass of `BetterTestCase`, and so forth.

- If given `Test0002a`, `JUnitSuperFinder` ascends the inheritance chain until it reaches either a matching superclass or `java.lang.Object`. It searches for the `TestCase` class by default, so when given `Test0002a`, it ascends two levels, finds `java.lang.Object`, and returns `Test0002a` to the harness as a test.
- If this test finder is given `java.util.ArrayList`, it ascends until it reaches `java.lang.Object`, at which point it decides that the class is not a test and moves on.

To change the superclass for which you are scanning, supply the `-superclass` argument and specify a class name. You can supply this argument multiple times to scan for multiple superclasses. For example, in the `testsuite.jtt` file you might specify the following:

```
finder = com.sun.javatest.junit.JUnitSuperTestFinder -superclass
         foo.BetterTestCase -superclass foo.CustomTestCase
```

Although it does not execute tests, the test finder attempts to pick out test methods by looking for public methods that begin with the string "`test`". It then lists these in a space-separated list, without the parameters (just the method name). The list might contain duplicates because the full signature is not evaluated. Semantics for this `TestDescription` value are loosely defined at this point. Public comment is welcome (submit your comments to the JT harness interest forum at `https://jtharness.dev.java.net`

This superclass finder generates the `TestDescription` (`com.sun.javatest.TestDescription`) values shown in TABLE 12-1.

TABLE 12-1 JUnitSuperTestFinder Test Description Values

Key	Value(s)
keywords	junit, junit3
`junit.finderscantype`	superclass
`junit.testmethods`	(list of identified test methods)

JUnitAnnotationTestFinder

This annotation test finder scans classes for the `org.junit.Test` annotation. It uses the same scanning strategy as `JUnitSuperTestFinder`.

This annotation finder generates the `TestDescription` (`com.sun.javatest.TestDescription`) values shown in TABLE 12-2TABLE 12-2.

TABLE 12-2 JUnitAnnotationTestFinder Test Description Values

Key	Value(s)
keywords	junit, junit4
junit.finderscantype	annotation
junit.testmethods	(list of identified test methods)

JUnitBareMultiTest

This is the execution class for JUnit 3.x style tests. Execution is accomplished using the class name supplied by the test finder (through the `TestDescription`) which is used to execute that class's `TestCase.runBare()` method. This might not be sufficient for all test suites. Output from `stdout` and `stderr` are captured. The test passes if no exceptions are thrown and fails if there are any `Throwable` results.

JUnitAnnotationMultiTest

This is the execution class for JUnit 4.x style tests. It takes the class that was identified by the test finder and executes it using the JUnit library `TestResult.Section` parts. Also, because execution is turned over to JUnit, it does not report any of its own debugging output during execution. (In the future, it would be useful to take more advantage of the Result API and any observer APIs that are available.)

Implementation Notes

The use of the `junit3` and `junit4` keywords might be a generalization, since it really represents how the class was found. A test suite might mix use of version 3 and 4 features, meaning it is not necessarily completely 4.x compliant. Nonetheless, perhaps being able to run 3.x style tests out of a mixed set (see `com.sun.javatest.finder.ChameleonTestFinder`) can be useful. Do not forget that the `junit` keyword is also added so that JUnit tests can be selected from among non-JUnit tests.

Two of the most likely changes you should make is to modify the test finder or modify how to execute the test. To change the test finder, simply subclass `JUnitTestFinder`, provide it on the class path in `testsuite.jtt` and change the finder setting in `testsuite.jtt`.

To change the method for executing a test, you must change how it is dispatched in `JUnitTestRunner`. To change that, you must subclass `JUnitTestRunner` and provide it on the `testsuite.jtt` class path. You must also subclass `JUnitTestSuite` and change its setting in `testsuite.jtt` (see).

Areas for Improvement

This section lists implementation features that might ben122efit from user feedback and further development. You can provide this on the JT harness web site (`https://jtharness.dev.java.net`).

- The use of class path is currently not convenient. The general design of the harness is that the setting in `testsuite.jtt` affects the tests, rather than the system class path that the harness uses. This area can be more refined.
- Some additional base implementations of the interview class would be useful. In particular, providing one that reads a properties file and dumps it directly into the `Map` of `Interview.export(Map)` would provide a "quick and dirty" way for people to configure their test suites. Perhaps the location of the file can be written as a setting in `testsuite.jtt`.

Note – Users should generally not be instructed to alter `testsuite.jtt`. These settings are designed to be static. Information the user provides should be gathered through the interview system. As an architect, you should configure the `testsuite.jtt` file for general use during the retrofit process. Once the conversion is completed, the file should remain relatively untouched.

- It might be useful to hard code the `Interview` class and accept an override in the `testsuite.jtt` file, rather than forcing the developer to specify it in the file as documented above. This also applies to the `JUnitTestRunner` (or just the `TestRunner` class) in the implementation of `JUnitTestSuite`.

References

- JT Harness project (`open source version of the JavaTest harness`)
- JUnit project (`http://SourceForge.net/projects/junit`)
- JUnit 3.X home page (`http://junit.sourceforge.net/junit3.8.1/index.html`)
- JUnit 4.X home page (`http://junit.sourceforge.net`)
  - API documentation (`http://junit.sourceforge.net/javadoc_40/index.html`)

- JUnit Cookbook
  (http://junit.sourceforge.net/doc/cookbook/cookbook/htm)

CHAPTER 13

Customization

This chapter describes customizations that can be made in the JavaTest harness appearance and function. Customizations not documented in this guide are described in the appropriate Javadoc tool (API) documentation.

While most of this guide describes customization, this chapter describes advanced customization beyond that usually required to configure and execute a test suite. In this chapter, topics discussed include:

For architects, this chapter is most useful either after you have developed a basic version of a test suite and want to customize harness capabilities or as an overview to see exactly how much of the harness can be customized.

Customization API

Several sections in this chapter refer to methods present in the ContextManager API with many of the harness customization features described in this chapter controlled by the `ContextManager` class (`com.sun.javatest.exec.ContextManager`). The harness queries this class to determine if a particular feature is enabled and if it is necessary for the supporting classes or objects to realize the customization.

An architect can create a single custom `ContextManager` implementation class for their test suite and override any methods as needed. The only thing required inside a custom implementation of `MyContextManager` is overriding the appropriate

methods for the customized features. No additional implementation is required beyond that provided by the base class. The base class is not abstract and the default implementation provides the default behavior intended for the harness.

Architects then provide the custom `ContextManager` class to the harness by adding a value as part of the `TestSuite` properties (see `TestSuite.getTestSuiteInfo(String)`). The property name that should be provided is `tmcontext`. This can be done programmatically in the `TestSuite` class, but is more easily accomplished by placing the value in the `testsuite.jtt` file (see Chapter 8) of the test suite. For example:

```
tmcontext=com.yourdomain.jtharness.MyContextManager
```

That class should be available to the harness in the classpath value that is also provided in the `testsuite.jtt` file. See Chapter 8 for more information about this file.

Note – The `classpath` value in `testsuite.jtt` is a space separated value.

Internationalization

Many harness APIs refer to resource bundles (`java.util.ResourceBundle`) and String keys instead of raw strings for presentation. For example, in the `JavaTestToolBar` API the default implementation of `getName()` and `getDescription()` make it easier to provide a resource bundle instead of overriding the methods. The API documentation for `JavaTestToolBar.getDescription()` states:

```
Get the long description of this toolbar's purpose.
May be multiple sentences if desired. This is automatically
retrieved from the supplied resource bundle by combining it with
the toolbar ID with getID(()), e.g. it will try to retrieve
getId().tb.desc from the resource bundle.
```

In the custom code that the test suite provides, there will be a line of code which creates the toolbar, `JavaTestToolBar`. The constructors for that class require passing a `ResourceBundle` and a `String` key.

```
public JavaTestToolBar(ResourceBundle bundle, String resourceID)
```

Assuming that the code creating the toolbar (provided by the test suite) is in package `foo.bar` and is called `MyContextManager`, it is common in the harness to write code similar to the following example:

```
package foo.bar;
import com.sun.javatest.util.I18NResourceBundle;
class MyContextManager extends ContextManager {
    ... {
       toolbar = new JavaTestToolBar(i18n, "mytoolbar");
    }
    private static I18NResourceBundle i18n =

I18NResourceBundle.getBundleForClass(MyContextManager.class);
}
```

This `i18n` object can then be reused throughout the lifetime of that custom context manager for any necessary purpose. In this case it is passed to the toolbar being customized.

Continuing with the code example and how `JavaTestToolBar` would use it, when `getDescription()` on that toolbar object is called by the harness, it attempts to retrieve `getId().tb.desc` from the resource bundle. In the example, the harness loads the string named `mytoolbar.tb.desc` from the file (on the classpath) `foo/bar/i18n.properties`. The content of `foo/bar/i18n.properties` might be:

```
# i18n file for package foo.bar
mytoolbar.tb.desc=This is my great toolbar for you to use.
mytoolbar.tb.name=My Tools
```

It is typical to provide each package with its own `i18n.properties` file and then use a single instance of `I18NResourceBundle` object within that package, passing it around as needed. See the `java.util.ResourceBundle` API documentation for more information about how it automatically resolves the name of the resource file to load and for the format of the entries in the bundle.

Customizing the Splash Screen

Instead of using the default JavaTest harness splash screen, architects can insert a custom test suite splash screen for users.

To use a custom splash screen, test suite architects must accomplish the following actions:

- Insert a `splash.properties` file that specifies the custom splash image in the test suite `lib/` directory (below `javatest.jtt`).

  Refer to "Example of `splash.properties` File" on page 126" for the content and format of a `splash.properties` file.

- Insert the splash image file in a location relative to the `splash.properties` file.

  Acceptable image formats for the splash screen are `GIF`, `JPEG` and `PNG`.

Once the `splash.properties` and the splash image files are integrated in the test suite `lib/` directory, the JavaTest harness will display the custom splash screen instead of the default when starting.

In the current implementation, the JavaTest harness displays the custom splash screen when users start the harness with `-ts` or `-testsuite` flags to specify the specific test suite. In the future additional flags might be used to start the harness with the custom splash screen.

Example of `splash.properties` File

The following is an example of the required format of the splash.properties file. In this example, the custom image name is `splashMyProduct.gif`.

```
# comment
splash.icon=splashMyProduct.gif
```

Notes About the Implementation

Because the splash screen must be capable of being internationalized, the `testsuite.jtt` file is not used to directly specify the splash screen. This capability of being internationalized requires that it should go through the standard ResourceBundle searching. The standard ResourceBundle searching is facilitated by using the `splash.properties` file. Other options for specifying the custom splash screen were not utilized because they increase the startup overhead of the harness by requiring it to perform additional file operations.

Customizing Menus

Test suite architects can customize the GUI menus in the Test Manager (ExecTool) by using the API provided by the harness. Common uses of this customization are to turn on and off frequently used test suite options or to trigger customized

informational dialogs. The API provides limited access to the menu structure of the Test Manager, but relatively unlimited capabilities for the menu items themselves (such as the ability to insert multi-level menus, enable or disable a menu item, or provide a checkbox menu item). In addition, the popup menu available on the main test tree can also be customized.

Adding Menu Items to Test Manager Menus

Menu additions are managed by the JavaTestMenuManager (`com.sun.javatest.exec.JavaTestMenuManager`) provided by the ContextManager (`com.sun.javatest.exec.ContextManager`). The menu manager class provides an abstract class for the architect to populate. The most important part of the class are the set of constants that it defines. These constants define a set of logical positions within the Test Manager's menu structure. Instead of allowing the architect to determine the exact position of the menu items, which makes it virtually impossible to make future harness menu changes, the API defines the logical positioning with which the harness will determine the final position of a custom menu item.

The constants take the form of <*logical menu*>_<*logical position*>, such as HELP_ABOUT. If the architect wanted to add a menu item labeled About My Test Suite, they would use the HELP_ABOUT menu position. In the same manner, for the FILE_OTHER position, the architect would use file related label or a related label that logically belongs in a location on the File menu.

See the JavaTestMenuManager API documentation provided by the harness for detailed implementation information.

Adding Menu Items to the Tree Popup Menu

Similar to customizing the main Test Manager menus, the ContextManager must provide a class to manage the popup menu items. However, unlike the JavaTestMenuManager, it does not manage the position of items or serve as a container of multiple menus. Instead, JavaTestContextMenu represents a single menu item (in the Swing sense) that is activated on demand. The class manages the underlying JMenuItem and the rules for displaying that item.

Note – This class assumes that the context menu is displayed in the context of a test folder or single test (such as a folder in the test tree or a test in the test tree). It cannot be used to insert context menus at other locations within the harness GUI and the availability of the custom menu items is limited to certain locations.

The most important considerations for this class are the rules for deciding whether or not to display the tree popup menu:

1) Is this menu item applicable to test entities, folders, or both?

2) Is the item applicable for cases in which multiple items (multiple tests for example) are selected?

For example, by using these rules, an architect can create popup menu items that appear only on tests, such as a menu item that says Configure Test. An architect can also create a menu item, such as Delete, that acts on homogeneous selection sets.

See the `JavaTestContextMenu` API documentation provided by the harness for more details.

Customizing Toolbars

The architect can add custom toolbars to the Test Manager, which are combined with the toolbars provided by the harness. The harness provides a default toolbar manager (`com.sun.javatest.exec.ToolBarManager`) which is suitable for most uses. Using either the default context manager or preferably a custom `ContextManager` (see "Customization API" on page 123), the toolbar manager is retrieved through the `getToolBarManager()` method. The API on this object allows the test suite to add and remove toolbars of the type `com.sun.javatest.exec.JavaTestToolBar`, which is a subclass of Swing's JToolBar.

On the `JavaTestToolBar` API, methods are provided that enable the harness to query the toolbar for its name, description and optional behavior. These methods enable the harness to automatically manage the toolbar, especially in the case of presenting menus which control visibility. Architects should pay attention to the internationalization practices that the harness enforces (see "Internationalization" on page 124). See the API documentation for `JavaTestToolBar` methods `getId()`, `getDescription()`, and `getName()`.

APPENDIX **A**

Standard Commands

The JavaTest harness provides a number of standard commands that you can use to configure an environment to run a test suite on your test platform. These commands all extend the standard JavaTest `Command` class.

With these standard commands, you can configure the JavaTest harness for a wide variety of test platforms. If, however, you find that you cannot create an environment for your test platform using these commands, you may need to write your own: see "Writing Custom Commands" on page 98 for more details.

The standard commands are as follows:

- ActiveAgentCommand: A command to execute a subcommand on a JavaTest Agent running in active mode
- ExecStdTestSameJVMCmd: A command to execute a simple API test in the same JVM in which the JavaTest harness or the JavaTest Agent is running
- ExecStdTestOtherJVMCmd: A command to execute a simple API test in a JVM that is separate from the JVM in which the JavaTest harness or the JavaTest Agent is running
- JavaCompileCommand: An example command that demonstrates how to invoke a Java application via a wrapper class
- PassiveAgentCommand: A command to execute a subcommand on a JavaTest Agent running in passive mode
- ProcessCommand: A command to execute a system command in a separate process
- SerialAgentCommand: A command to execute a subcommand on a JavaTest Agent, communicating via a serial line

Note – Examples in this appendix use Unix style commands and file separators.

ActiveAgentCommand

A command to execute a command in a separate JVM, typically on a remote machine, by delegating it to a JavaTest Agent which has been configured to run in active mode. This means it contacts the JavaTest harness to determine what it should do.

The JavaTest active agent pool must be started before you start running tests that use this command. The active agent pool holds the requests from the active agents until they are required. You can start the active agent pool from the JavaTest GUI or command line.

Usage	`com.sun.javatest.agent.ActiveAgentCommand` [*options*] *command-class* [*command-arguments*]
Arguments Options	**Description**
`-classpath` path	This option allows you to specify a classpath *on the system running the JavaTest harness* from which to load the command class and any classes it invokes. The classes are automatically loaded into the agent as needed. If the class path is not specified, the classes are loaded from the agent's class path. See Chapter 4 for additional information about using the `-classpath` option.
`–mapArgs`	The command to be executed might contain values that are specific to the host running the JavaTest harness and that might not be appropriate for the host that actually runs the command. If this option is given, the agent uses a local mapping file to translate specified string values into replacement values. This is typically used to map filenames from the view on one host to the view on another. See the JavaTest online help for more information.
`–tag` *tag*	This option allows the user to specify a string that is used to identify the request on the agent. If not specified, the default value, *command-class*, is used. It is suggested that the URL of the test should be used as the value for this option. A configuration can use the symbolic name `$testURL`, which is substituted when the command is executed.
`command` *class*	The name of a command class to be executed by the agent. If the -classpath option is not used, the class should be on the classpath of the agent, and should be appropriate for the agent, depending on the security restrictions in effect. For example, an agent running as an application might be able to run a `ProcessCommand`, but an agent running as an applet might not. The class should implement the interface `com.sun.javatest.Command`.

`command` *arguments*

The arguments to be passed to the run method of an instance of the command class running on the agent. The arguments can be translated to agent-specific values if the `-mapArgs` option is given.

Description

`ActiveAgentCommand` is a facility to execute a command on a JavaTest Agent that has been configured to run in active mode. A JavaTest Agent provides the ability to run tests in a context that might not be able to support the JavaTest harness. This could be because the tests are to be run on a machine with limited resources (such as memory), or in a security-restricted environment (such as a browser), or on a newly developed platform on which it is not possible to run the JDK.

Commands often contain host-specific arguments, such as filenames or directories. Although the files and directories might be accessible from the agent host (and in general, should be), the paths might be different. For example, `/usr/local` on a Solaris platform might be mounted as a network drive like `H:` on a Windows platform. When an agent is initialized, it may be given information on how to translate strings from one domain to another. On a per-command basis, the agent can be instructed to translate a command according to the translation tables it is given.

The command to be executed on an agent can be identified with a tag for tracing and debugging purposes. If none is specified, a default identification is used.

Any output written by the command when it is executed by the agent appears as the output of the `ActiveAgentCommand` command itself. If the command is successfully executed by the agent (i.e. the `Command` object is successfully created and the `run` method invoked), the result of `ActiveAgentCommand` is the result of the command executed by the agent. Otherwise, an appropriate error status is returned.

Examples

Using `ActiveAgentCommand` to Execute a `ProcessCommand` on an Active Agent:

This example is based on the following sample code demonstrating `ProcessCommand`:

```
com.sun.javatest.lib.ProcessCommand
/usr/local/jdk1.3/solaris/bin/javac
     -classpath /home/juser/classes -d /home/juser/classes
HelloTest.java
```

To make a command execute on another machine, prefix it with `ActiveAgentCommand` and any arguments that `ActiveAgentCommand` requires:

```
compile.java=com.sun.javatest.agent.ActiveAgentCommand
com.sun.javatest.lib.ProcessCommand \
     /usr/local/jdk1.3/solaris/bin/javac \
     -classpath /home/juser/classes \
     -d /home/juser/classes HelloTest.java
```

See Also

All the other standard commands in this appendix. Subject to security restrictions on the agent, they can all be executed remotely by means of `ActiveAgentCommand`.

ExecStdTestSameJVMCmd

A command that executes a standard test in the same JVM in which JavaTest Agent is running.

Usage `com.sun.javatest.lib.ExecStdTestSameJVMCmd` [*options*] *test_class* [*test_args*]

Arguments *Options*

–loadDir *directory*

Creates a `ClassLoader` that loads any necessary classes from the specified directory. The `ClassLoader` is garbage collected once `ExecStdTestSameJVMCmd` has completed. If you do not specify `-loadDir`, the system class loader is used. Using a separate `ClassLoader` for each test reduces the chance that one test interferes with another. Also, using a separate `ClassLoader` allows the command to unload test classes after the test is executed, which could be critical in memory constrained environments.

On some systems, the security manager restricts the ability to create a `ClassLoader`. If you use this option and cannot create a `ClassLoader`, the command throws a `SecurityException`.

test class

Specifies the name of the test class to execute. This class must be a subtype of `com.sun.javatest.Test`. To specify a class in the test description currently being processed by the JavaTest harness, use the `$executeClass` substitution variable.

test args

Specifies a list of arguments to be passed to the `run` method of the class being executed. To specify arguments in the test description currently being processed by the JavaTest harness, use the `$executeArgs` substitution variable

Description `ExecStdTestSameJVMCmd` is a JavaTest command that executes a standard test in the same JVM in which the JavaTest Agent is running. The class must be a subtype of `com.sun.javatest.Test`.

`ExecStdTestSameJVMCmd` creates a new instance of the class, calls its `run` method, and passed the class args. If the class is successfully created and invoked, the result of `ExecStdTestSameJVMCmd` is equal to the result of the `run` method of the object.

Examples **Simple use of ExecStdTestSameJVMCmd:**

```
command.execute=com.sun.javatest.lib.ExecStdTestSameJVMCmd \
     $testExecuteClass $testExecuteArgs
```

Using ExecStdTestSameJVMCmd Inside an Environment:

```
com.sun.javatest.lib.ExecStdTestSameJVMCmd HelloTest
```

See Also `ExecStdTestOtherJVMCmd`

ExecStdTestOtherJVMCmd

A variant of ProcessCommand that executes a standard test using a subcommand in a separate process.

Usage

`com.sun.javatest.lib.ExecStdTestOtherJVMCmd` [*options*]
[*shell variables*] `subcommand` [*args*]

Arguments

Options

–v

Used for verbose mode. When `ExecStdTestOtherJVMCmd` is in verbose mode, additional output information is sent to the TestResult object.

shell variables

Specifies one or more shell environment values that are required by the sub-command. Shell environment variables are written as: name=value.

subcommands

Specifies the name of a program that is executed.

args

Specifies the arguments that are passed to the subcommand.

Description

`ExecStdTestOtherJVMCmd` is a JavaTest command that executes a test with a subcommand in a separate process (using a separate runtime). You would normally use this command to invoke a JVM to run the test class.

Examples of subcommands are the compiler for the Java programming language (javac) and the JVM (java). Normally, a test exits by creating a Status object and then invoking its exit method. This command also returns a Status object, which is equal to the object returned by the test.

Examples

Simple Use of ExecStdTestOtherJVMCmd

```
com.sun.javatest.lib.ExecStdTestOtherJVMCmd \
     /usr/local/jdk1.4/solaris/bin/java \
     -classpath /home/juser/classes
    HelloTest
```

Using ExecStdTestOtherJVMCmd Inside an Environment

```
command.execute=com.sun.javatest.lib.ExecStdTestOtherJVMCmd \
     /usr/local/jdk1.4/solaris/bin/java \
     -classpath /home/juser/classes
      $testExecuteClass $testExecuteArgs
```

See Also

`ExecStdTestSameJVMCmd`

`ProcessCommand`

JavaCompileCommand

Invokes a compiler in the same JVM in which the JavaTest harness or the JavaTest Agent is running.

Usage

com.sun.javatest.lib.JavaCompileCommand

```
[-compiler compiler-spec] [args]
```

Arguments

Options

`-compiler` *compiler-spec*

If the `-compiler` option is given, compiler-spec specifies the class name for the compiler, optionally preceded by a name for the compiler followed by a ":". If no compiler name is given before the class name, the default name is "java" followed by a space and then the class name. If the `-compiler` option is not given, the default value for compiler-spec is `javac:sun.tools.javac.Main`.

args

Specifies the arguments to the compiler's compile method. If you use the default compiler, javac, the arguments are exactly the same as those you would use for javac. In this case, you should refer to documentation for the JDK for more details. Otherwise, refer to the documentation for the compiler you specify.

Description

This command is primarily an example that shows how any application written in the Java programming language can be interfaced to the JavaTest harness by writing a wrapper command. By default, the application in this example is the JDK compiler, javac, but any class implementing the same signature can be invoked. javac is normally run from the command line, per its specification, but it does have an undocumented interface API, that is sufficiently typical to be used as the basis for this example.

The compiler is assumed to have a constructor and compile method matching the following signature:

```
public class COMPILER {
     public COMPILER(java.io.OutputStream out, String name);
     boolean compile(String[] args);
}
```

When `JavaCompileCommand` is used, an instance of the compiler is created. The constructor is passed a stream to which to write any messages, and the name of the compiler to be used in those messages. Then, the compile method is called with any args passed to `JavaCompileCommand`. If the compile method returns true, the result is a status of "passed"; if it returns false, the result is "failed". If any problems arise, the result is "error".

The source code for this example is provided in the examples directory. It is the file `JavaCompileCommand.java` in the directory `src/share/classes/com/sun/javatest/lib/` under the main JavaTest installation directory.

Examples

Simple use of `JavaCompileCommand`

```
com.sun.javatest.lib.JavaCompileCommand HelloWorld.java
```

Using `JavaCompileCommand` **Inside an Environment**

```
command.compile.java=com.sun.javatest.lib.JavaCompileCommand \
    -d $testClassDir $testSource
```

Using `JavaCompileCommand` **to Invoke the RMI compiler**

```
command.compile.java=com.sun.javatest.lib.JavaCompileCommand \
     -compiler rmic:sun.rmi.rmic.Main \
     -d $testClassDir $testSource
```

See Also `ProcessCommand`

PassiveAgentCommand

A command to execute a command on a remote machine by delegating it to a JavaTest Agent that is configured to run in passive mode.

Usage `com.sun.javatest.agent.PassiveAgentCommand` [*options*] *command* *-class* [*command-arguments*]

Arguments *Options*

`-classpath` *path*

This option lets you to specify a classpath *on the system running the JavaTest harness* from which to load the command class and any classes it invokes. The classes are automatically loaded into the agent as needed. Otherwise, classes are loaded using the agent's class path. See Chapter 4 for additional information about using the `-classpath` option.

`–host` *host-name*

Specifies the host on which to run the command. A passive JavaTest Agent must be running on this host to execute the command. The option must be given; there is no default.

`–mapArgs`

The command to be executed might contain values that are specific to the host running the JavaTest harness and that might not be appropriate for the host that actually runs the command. If this option is given, the agent uses a local mapping file to translate specified string values into replacement values. This is typically used to map filenames from the view on one host to the view on another. See the JavaTest online help for more information.

`–port` *port-number*

This option specifies the port to which to connect when requesting an agent to run a command. The default value, 1908, is used if no value is explicitly given.

–tag *tag*

This option lets the user specify a string that identifies the request on the agent. If not specified, the default value, *command-class*, is used. It is suggested that the URL of the test be used as the value for this option. A configuration can use the symbolic name `$testURL`, which is substituted when the command is executed.

command class

The name of a command class to be executed by the agent. The class should be on the classpath of the agent and be appropriate for the agent, depending on the security restrictions imposed on the agent. For example, an agent running as an application might be able to run a `ProcessCommand`, but an agent running as an applet might not. The class should implement the standard interface `com.sun.javatest.Command`.

command args

The arguments to be passed to the run method of an instance of the command class running on the agent. The arguments might be translated to agent-specific values if the `–mapArgs` option is given.

Description

`PassiveAgentCommand` is a facility to execute a command on a JavaTest Agent that has been configured to run in passive mode. A JavaTest Agent provides the ability to run tests in a context that might not be able to support the entire JavaTest harness. Factors that require use of the JavaTest Agent include limited resources (such as memory), or in a security-restricted environment (such as a browser), or on a newly developed platform on which is not possible to run the JDK.

The host and port options identify an agent to be used to execute the command. The JavaTest harness attempts to contact an agent on that system that is running and waiting for requests.

Commands often contain host-specific arguments, such as filenames or directories. Although the files and directories might be accessible from the agent host (and in general, should be), the paths might be different. For example, `/usr/local` on a Solaris platform can be mounted as a network drive like `H:` on a Windows NT platform. When an agent is initialized, it may be given information on how to translate strings from one domain to another. On a per-command basis, the agent can be instructed to translate a command according to the translation tables it is given.

The command to be executed on an agent can be identified with a tag for tracing and debugging purposes. If none is specified, a default identification is used.

Any output written by the command when it is executed by the agent appears as the output of the `PassiveAgentCommand` command itself. If the command is successfully executed by the agent (i.e. the `Command` object is successfully created and the `run` method invoked) then the result of `PassiveAgentCommand` is the result of the command executed by the agent. Otherwise, an appropriate error status is returned.

Examples

Using `ActiveAgentCommand` to Execute a `ProcessCommand` on an Active Agent:

```
compile.java=\
     com.sun.javatest.agent.PassiveAgentCommand -host calloway \
     com.sun.javatest.lib.ProcessCommand \
     /usr/local/jdk1.3/solaris/bin/javac \
     -classpath /home/juser/classes \
     -d /home/juser/classes HelloTest.java
```

See Also

All the other standard commands in this appendix. Subject to security restrictions on the agent, they can all be executed remotely by means of `PassiveAgentCommand`.

ProcessCommand

Usage

`com.sun.javatest.lib.ProcessCommand` [*options*] [*env variables*] *command* [*command-arguments*]

Arguments

Options

`-v`

Verbose mode: tracing information is output to the log.

env variables

This is a list of named values to be passed as environment variables to the command to be executed. Each named value should be written as *name=value*.

command

This is the name of the command to be executed in a separate process.

command arguments

This is a list of arguments to be passed to the command to be executed.

Description

`ProcessCommand` executes a system command in a separate process with the specified set of environment variables and arguments.

The result of the command is a Status object based upon the exit code of the process. An exit code of zero is interpreted as `Status.PASSED`; all other exit codes are interpreted as `Status.FAILED`. There are variants of `ProcessCommand` that provide different interpretations of the exit code. These variants can be used in more specialized circumstances, such as running tests that use exit codes like 95, 96, and 97.

`ProcessCommand` copies the standard output stream of the process to the out2 command stream, and the standard error stream of the process to the out1 command stream.

Examples

Simple use of `ProcessCommand`

```
com.sun.javatest.lib.ProcessCommand
/usr/local/jdk1.3/solaris/bin/javac
-classpath /home/juser/classes -d /home/juser/classes HelloTest.java
```

Using `ProcessCommand` in an Environment

```
compile.java=com.sun.javatest.lib.ProcessCommand \
      /usr/local/jdk1.3/solaris/bin/javac \
      -classpath /home/juser/classes \
      -d /home/juser/classes $testSource
```

See Also

`ExecStdTestOtherJVMCmd`

SerialAgentCommand

A command to execute a command on a remote machine, by delegating it to a JavaTest Agent that has been configured to communicate via a serial RS232 line.

Usage com.sun.javatest.agent.SerialAgentCommand [*options*] *command-class* [*command-arguments*]

Arguments *Options*

-classpath *path*

This option lets you specify a classpath *on the system running the JavaTest harness* from which to load the command class and any classes it invokes. The classes are automatically loaded into the agent as needed. See Chapter 4 for additional information about using the `-classpath` option.

–mapArgs

The command to be executed might contain values that are specific to the host running the JavaTest harness and that might not be appropriate for the host that actually runs the command. If this option is given, the agent uses a local mapping file to translate specified string values into replacement values. This is typically used to map filenames from the view on one host to the view on another. See the JavaTest online help for more information.

–port *port-name*

This option specifies the name of the serial port on the system running the JavaTest harness to be used to communicate with a JavaTest Agent that has also been configured to communicate via a serial line. The set of possible names is determined dynamically, and is dependent on the underlying implementation of the `javax.comm` API. On Solaris, the names are typically *ttya*, *ttyb*; on a PC, the names are typically *COM1*, *COM2*, *COM3* and *COM4*.

–tag *tag*

This option lets the user specify a string to be used to identify the request on the agent. If not specified, the default value, *command-class*, is used. It is suggested that the URL of the test be used as the value for this option. In an environment file, this is available as the symbolic name "`$testURL`".

command class

The name of a command class to be executed by the agent. The class should be on the class path of the agent, and should be appropriate for the agent, depending on the security restrictions imposed on the agent. For example, an agent running as an application might be able to run a `ProcessCommand`, but an agent running as an applet might not.

command arguments

The arguments to be passed to the run method of an instance of the command class running on the agent. The arguments can be translated to agent-specific values if the `–mapArgs` option is given.

Description `SerialAgentCommand` is a facility to execute a command on a JavaTest Agent that has been configured to communicate via a serial line. A JavaTest Agent lets you run tests in a context that might not be able to support all of the JavaTest harness. This might be because the tests are to be run on a machine with limited resources (such as memory), or in a security-restricted environment (such as a browser), or on a newly developed platform on which is not possible to run the JDK.

The port option identifies a serial port on the system running the JavaTest harness, which should be connected to a serial port on the system running the JavaTest Agent. The serial line is accessed via the `javax.comm` optional package. This is not part of the standard JDK, and must be added to your class path when you start the JavaTest harness.

Commands often contain host-specific arguments, such as filenames or directories. Although the files and directories might be accessible from the agent host (and in general, should be), the paths might be different. For example, `/usr/local` on a Solaris platform could be mounted as a network drive like `H:` on a Windows NT platform. When an agent is initialized, it may be given information on how to translate strings from one domain to another. On a per-command basis, the agent can be instructed to translate a command according to the translation tables it is given.

The command to be executed on an agent can be identified with a tag for tracing and debugging purposes. If none is specified, a default identification is used.

Any output written by the command when it is executed by the agent appears as the output of the `SerialAgentCommand` command itself. If the command is successfully executed by the agent (i.e. the `Command` object is successfully created and the `run` method invoked), then the result of `SerialAgentCommand` is the result of the command executed by the agent. Otherwise, an appropriate error status is returned.

Examples This example is based on the following sample code demonstrating `ProcessCommand`:

```
com.sun.javatest.lib.ProcessCommand /usr/local/jdk1.3/solaris/bin/javac
     -classpath /home/juser/classes -d /home/juser/classes HelloTest.java
```

A command can be made to execute on another machine simply by prefixing it with `SerialAgentCommand` and any arguments that `SerialAgentCommand` requires.

```
compile.java=\
     com.sun.javatest.agent.SerialAgentCommand -port ttya \
     com.sun.javatest.lib.ProcessCommand \
     /usr/local/jdk1.3/solaris/bin/javac -classpath /home/juser/classes
     -d /home/juser/classes HelloTest.java
```

See Also All the other standard commands in this appendix. Subject to security restrictions on the agent, they can all be executed remotely by means of `SerialAgentCommand`.

APPENDIX **B**

Formats and Specifications

This appendix describes file formats and specifications that test architects should know about.

Test URL Specification

This specification describes how test files must be named to work properly with the JavaTest harness.

The JavaTest harness converts the native path names of tests to an internal format called a *test URL*. When the JavaTest harness converts a path name to a test URL, it:

- Makes the path relative to the root of the test suite
- Converts the path separator to a forward slash

A test URL consists of three components:

- The folder/directory path name (relative to the test suite `tests` directory)
- The name of the file that contains the test description
- An optional *test identifier* can be appended to designate a test description table within a test description file

For example:

```
api/javatest/TestSuite.html#getName
```

The JavaTest harness evaluates test URLs without regard to case; however, it does preserve case when possible to increase readability for the user.

The path names you create can contain only the following characters:

- ISO Latin 1 characters A-Z and a-z
- Digits 0-9

- Underscore character "_"
- Dash character "-"
- Period character "." (*deprecated*)
- Open and close parentheses (*deprecated*)

Test URLs must never contain whitespace characters.

Deprecated characters are included for backward compatibility; please do not use them as they might become unsupported in future releases. Whenever possible, use short names to make best use of screen real estate.

Note – When the result file (`.jtr`) is created, the text after the last period is omitted.

The test identifier may only contain the following characters:

- ISO Latin 1 characters A-Z and a-z
- Digits 0-9
- Underscore character "_"

Test Paths

An *test path* can be used by a user to specify a particular subset of tests in the test suite hierarchy; for example, in the Tests tab of the JavaTest configuration editor (Standard Values view).

Initial URLs specify a set of tests in the following ways:

- A folder that contains one or more tests
- A file that contains one or more tests
- A single, complete test URL

The test path conforms to the rules specified in the previous section, but is not required to resolve the complete URL. If the test path is an incomplete test URL (for example, a folder), the JavaTest harness generates a list of the tests' URLs contained hierarchically beneath it.

Exclude List File Format

Test suites use the exclude list mechanism to identify tests that should not be run. The JavaTest harness consults the exclude list when it selects tests to run, and does not run the tests on the list. Excluded tests normally appear as *filtered out* in the JavaTest test tree.

When the JavaTest harness is used with TCK test suites, the exclude list mechanism is used to determine the correct set of tests that must be executed for certification. The exclude list mechanism is a mechanism for "removing" broken or invalid tests in the field without having to ship a new test suite.

Syntax

The exclude list is a four-column table that uses ISO Latin 1 (ISO 8859-1) character encoding. Lines that are completely empty or contain only whitespace (space, tab, newline) are allowed. Comment lines begin with the "#" character. Each line has the following format:

Test_URL[*Test_Cases*] *BugID_List Keywords Synopsis*

For example:

api/index.html#attributes[Char2067] 4758373 reference,test Bug is intermittent

TABLE B-1 Exclude List Field Descriptions

Field	Description
Test_URL[Test_Cases]	The URL of the test to be excluded. This field can specify an entire test or can use the Test_Cases parameter to specify a subset of its test cases. The test cases field is optional and the brackets must not be present if the field is empty.
BugID_List	A comma-separated (no spaces) list of bug identifiers associated with the excluded test.
Keywords	A comma-separated (no spaces) list of keywords that can be used to classify exclusions. The particular values are project specific.
Synopsis	A short comment that describes the reason the test is excluded. This optional field is recommended because it helps track entries on the exclude list.

Each field is separated by spaces and/or tabs. A line break terminates the entry. There is no way to indicate that the entry continues on the next line. Comments can appear on any line of the file, see the rules below.

Although it is not recommended, you can omit the synopsis, keywords, or bugID_List field; however, the entry is only valid if everything to the right of the omitted field is also omitted. For example, you cannot omit a bugID and include a keyword; but you can include a bugID and omit the keywords and synopsis.

Test URL and Test Cases

Entries must not specify overlapping test cases. For example, you cannot exclude an entire test and then exclude a test case inside that test. These two entries can never appear in the same file:

```
api/java_lang/Character/index.html#attributesFullRange
api/java_lang/Character/index.html#attributesFullRange[Character2067
]
```

The URL must specify a specific test or test case. Entire subsections of the test suite cannot be excluded with a single entry. Continuing with the API example, if a test suite is rooted at the *ts_dir*`\tests` directory and the `index.html` file contains many test descriptions, all of the following test URLs are invalid:

```
api
api/java_lang/
api/java_lang/Character
api/java_lang/Character/index.html
tests/api/java_lang/xyz
java_lang/xyz
```

You can exclude individual test cases within a test description by appending a list of those tests cases at the end of the test URL. The list of test cases must be enclosed within square brackets. The list of test cases is separated by commas with no internal whitespace. There is no whitespace between the end of the test URL and the opening square brackets. The following figure shows valid test URL entries:

```
vm/instr/ifnull/ifnull003/ifnull00303m1/ifnull00303m1.html
api/java_beans/beancontext/BeanContextMembershipEvent/index.html#Constructor
api/java_lang/Character/index.html#attributesFullRange[Character2067]
api/SystemFlavorMap/index.html#method[SystemFlavorMap0001,SystemFlavorMap0004]
```

For information about constructing valid test URLs, see "Test URL Specification" on page 141.

BugIDs

The list of bug IDs is separated by commas, and contains no whitespace characters. Items in the BugID_List are entries that correspond to a bug in a bug tracking system. Letters, integers, dashes and underscore characters are valid.

Keywords

It is recommended that keyword entries be no longer than 20 characters long. Keyword entries must start with a letter and can contain letters, numbers, and the underscore character. The list of keywords is separated by commas, without any whitespace characters.

Synopsis

Any description or notes about a particular entry. There are no special restrictions on the content of this field. It is recommended that the field contain no more than 100 characters.

Comments and Header Information

Comments always extend from column zero of a line to end of the line. To be a comment line, the character in column zero must be "#"; two consecutive "#" characters at the beginning of a line are allowed, but the use of three or more is reserved.

Optional (but recommended) header lines can be added to your exclude list file to improve readability. Header lines always begin with "###" and can be used to designate titles, and revision histories. The format is:

header_type heading content...

The case-sensitive header type specification is separated from the "###" prefix by white space characters, the heading content is separated from the header type specification by more whitespace characters. These values should appear only once in any exclude list file, and it is recommended that they be placed at the top of the file. Currently, the only supported header type is "title". The title describes the exclude list and must be terminated with a newline.

The following is an example of a valid exclude list:

```
### title My example exclude list
### revised Mon Jul 23 18:15:04 PDT 2001
api/java_lang/runtimetest.java
```

```
# this is a comment line
api/index.html#attributes[Char2067] 1234567 reference,test
Invalid assumption
## this is another comment line
api/mytest.java#1 1234568,987654321 spec
```

APPENDIX C

What Technical Writers Should Know About Configuration Interviews

Technical writers can greatly contribute to the quality of a JavaTest configuration interview — think of the text in a configuration interview as being equivalent to an application's user interface; the better the text, the easier the test suite is to run. There are two areas where a writer's contribution is extremely important:

- The careful construction and phrasing of the question text
- Providing extra help and examples in the More Info pane

Question Text

Interview questions should be written as concisely, consistently, and clearly as possible. Any amplification, clarification, or examples should be included in the More Info pane.

Not all questions are really questions; some "questions" are written as statements that instruct the user to specify, choose, or select an item of information.

To see an example interview, run the JavaTest tutorial in Chapter 2. The tutorial uses the Demo TCK that is part of the JavaTest Architect's release.

Question text is kept in a Java properties file associated with the interview class files; you get the path to the properties file from the interview developer. Every interview question is identified by a unique key. The key is based on a name assigned by the developer and should uniquely identify the question with the interview. Keys are of the form:

interview_class_name.*question_name*

The following is a snippet from the Demo TCK interview properties file:

```
title=Demo Interview Configuration Editor
.
.
DemoTCKParameters.cmdType.smry=How to Run Tests
using a JavaTest Agent?computer (using a separate JVM), or to run
them on another computer DemoTCKParameters.cmdType.text=Do you
wish to run the tests on this
DemoTCKParameters.cmdType.agent=Using a JavaTest Agent
DemoTCKParameters.cmdType.otherVM=On this computer
DemoTCKParameters.data.smry=Test Configuration Values...
local settings of some parameters required by some of the
tests.DemoTCKParameters.data.text=The following questions
determine the
DemoTCKParameters.desc.smry=Description
identify the configuration you are creating
here.DemoTCKParameters.desc.text=Please provide a short
description to
.
.
```

The file contains the following types of elements:

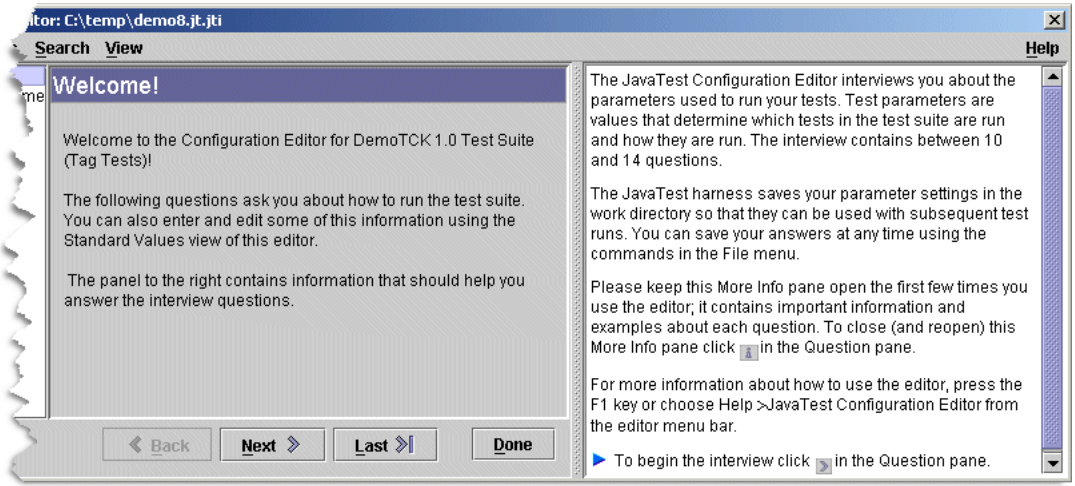
- The title of the full interview
- A title for each question of the form: *question_key*`.smry`
- The text for each question of the form: *question_key*`.text`
- Additional entries for choice items that are localized

Note – Do not type carriage return or linefeed characters within entries in a Java properties files. This causes an error when the interview is built. Use the "`\n`" characters to specify line breaks in question text.

More Info

As a technical writer, you can really add value to a test suite in the configuration interview More Info pane.

FIGURE C-1 The JavaTest Configuration Editor: Question and More Info Panes



The text displayed in the More Info pane is formatted using HTML 3.2 tags and provides a reasonably rich environment for formatting text. Although the text can be created using a WYSIWYG HTML editor, most More Info text is short and simple and is easy to create manually using a text editor.

Typically, the developer who creates the configuration interview creates the basic More Info system, seeding it with empty HTML files that you can fill in with text.

Experience has shown that it is best to create one HTML More Info file per interview question. It is also a good idea to name the More Info HTML files after the property names of the questions they describe. For example, in the snippet in the previous section, you can see that the `DemoTCKParameters` interview contains a question named `cmdType` — you should expect to see a corresponding More Info file named `cmdType.html`.

Formatting Styles

The following tips can be helpful when formatting your More Info topics:

Your More Info topics should link to the style sheet included in this package named `moreInfo.css`.

- Use only HTML 3.2 tags because of limitations in the HTML viewer
- Section 508 accessibility rules may apply
- Do not create hypertext links to other More Info topics or other external documents
- Do not add any `<p>` tags between the `<body>` tag and the *initial* paragraph

- Use the <i> tag for variables in command lines — use the tag for emphasis
- All file names and code examples should be formatted in the fixed-width font using the <code> or <pre> tags
- The style sheet (moreInfo.css) contains a "tight" class that you can use with <li> tags for lists in which you want less vertical space between list items. For example:
- <li class="tight">This item is closer to the previous list item
- Indent path names and code examples using the <p class="code"> tag. The code class indents the lines 8 pixels and has no top margin. For example, the following HTML:

```
<p class="code">
   <i>jdk_install_dir</i><code>/bin/java</code>
</p>
<p class="code">
   <i>jdk_install_dir</i><code>/jre/java</code>
</p>
```

Produces the following output:

jdk_install_dir/bin/java
jdk_install_dir/jre/java

Usage and Conventions

The following list describes some conventions that have proven useful for writing More Info text:

- Use the present tense when possible. For example, instead of:

  "The following questions will gather..."

  use:

  "The following questions gather..."
- When reasonable, provide examples in both Unix and Microsoft Windows format

Glossary

A

active agent — A type of test agent that initiates a connection to the JavaTest harness. Active test agents allow you to run tests in parallel using many agents at once and to specify the test machines at the time you run the tests. Use the agent monitor to view the list of registered active agents and synchronize active agents with the JavaTest harness before running tests. See also test agent, passive agent, and JavaTest agent.

agent monitor — The JavaTest window that is used to synchronize active agents and to monitor agent activity. The Agent Monitor window displays the agent pool and the agents currently in use.

agents — See test agent, active agent, passive agent, and JavaTest agent.

Application Programming Interface (API) — An API defines calling conventions by which an application program accesses the operating system and other services.

assertion — A statement contained in a structured Java technology API specification to specify some necessary aspect of the API. Assertions are statements of required behavior, either positive or negative, that are made within the Java technology specification.

assertion testing — Compatibility testing based on testing assertions in a specification.

atomic operation — An operation that either completes in its entirety (if the operation succeeds) or no part of the operation completes at all (if the operation fails).

B

behavior-based testing A set of test development methodologies that are based on the description, behavior, or requirements of the system under test, not the structure of that system. This is commonly known as "black-box" testing.

C

class The prototype for an object in an object-oriented language. A class may also be considered a set of objects which share a common structure and behavior. The structure of a class is determined by the class variables which represent the state of an object of that class and the behavior is given by a set of methods associated with the class. See also classes.

classes Classes are related in a class hierarchy. One class may be a specialization (a "subclass") of another (one of its "superclasses"), it may be composed of other classes, or it may use other classes in a client-server relationship. See also class.

compatibility rules Compatibility rules define the criteria a Java technology implementation must meet in order to be certified as "compatible" with the technology specification. See also compatibility testing.

compatibility testing The process of testing an implementation to make sure it is compatible with the corresponding Java technology specification. A suite of tests contained in a Technology Compatibility Kit (TCK) is typically used to test that the implementation meets and passes all of the compatibility rules of that specification.

configuration Information about your computing environment required to execute a Technology Compatibility Kit (TCK) test suite. The JavaTest harness version 3.x uses a configuration interview to collect and store configuration information. The JavaTest harness version 2.x uses environment files and parameter files to obtain configuration data.

configuration editor The dialog box used JavaTest harness version 3.x to present the configuration interview.

configuration interview A series of questions displayed by JavaTest harness version 3.x to gather information from the user about the computing environment in which the TCK is being run. This information is used to produce a test environment that the JavaTest harness uses to execute tests.

configuration value Information about your computing environment required to execute a TCK test or tests. The test environment version 3.x uses a configuration interview to collect configuration values. The JavaTest harness version 2.x uses environment files and parameter files to obtain configuration data.

E

equivalence class partitioning A test case development technique which entails breaking a large number of test cases into smaller subsets with each subset representing an equivalent category of test cases.

exclude list A list of TCK tests that a technology implementation is not required to pass in order to certify compatibility. The test environment uses exclude list files (`*.jtx`), to filter out of a test run those tests that do not have to be passed. The exclude list provides a level playing field for all implementors by ensuring that when a test is determined to be invalid, then no implementation is required to pass it. Exclude lists are maintained by the Maintenance Lead (ML) and are made available to all technology licensees. The ML may add tests to the exclude list for the test suite as needed at any time. An updated exclude list replaces any previous exclude lists for that test suite.

H

HTML test description A test description that is embodied in an HTML table in a file separate from the test source file.

I

implementation See technology implementation.

instantiation In object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

J

Java Platform, Standard Edition (Java SE) The Java SE platform is a set of specifications that defines the desktop runtime environment required for the deployment of Java applications. Java SE technology implementations are available for a variety of platforms, but most notably Solaris and Microsoft Windows.

Java Application Manager (JAM) A native application used to download, store and execute Java applications.

Java Archive (JAR) A JAR is a platform-independent file format that combines many files into one.

Java Platform Libraries The class libraries that are defined for each particular version of a Java technology in its Java technology specification.

Java technology A Java technology is defined as a Java technology specification and its reference implementation (RI). Examples of Java technologies are Java Platform, Standard Edition (Java SE), the Connected Limited Device Configuration (CLDC), and the Mobile Information Device Profile (MIDP).

Java technology specification A written specification for some aspect of the Java technology.

JavaTest agent A test agent supplied with the JavaTest harness to run TCK tests on a Java implementation where it is not possible or desirable to run the main JavaTest harness. See also test agent, active agent, and passive agent.

JavaTest harness The JavaTest Harness is a test harness that has been developed to manage test execution and result reporting for a Technology Compatibility Kit (TCK). The harness configures, sequences, and runs test suites. The JavaTest harness is designed to provide flexible and customizable test execution. It includes everything a test architect needs to design and implement tests for Java technology specifications.

K

keywords Keywords are defined for a test suite by the test suite architect. Keywords are used to direct the JavaTest harness to include or exclude tests from a test run.

M

Maintenance Lead (ML) The person responsible for maintaining an existing Java technology specification and related reference implementation (RI) and Technology Compatibility Kit (TCK). The ML manages the TCK appeals process, exclude list, and any revisions needed to the specification, TCK, or RI.

methods Procedures or routines associated with one or more classes, in object-oriented languages.

MultiTest A JavaTest library class that enables tests to include multiple test cases. Each test case can be addressed individually in a test suite exclude list.

N

namespace A set of names in which all names are unique.

O

object-oriented A category of programming languages and techniques based on the concept of objects which are data structures encapsulated with a set of routines, called methods, which operate on the data.

objects In object-oriented programming, objects are unique instances of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

P

packages A namespace within the Java programming language. It can have classes and interfaces. A package is the smallest unit within the Java programming language.

passive agent A type of test agent that must wait for a request from the JavaTest harness before they can run tests. The JavaTest harness initiates connections to passive agents as needed. See also test agent, active agent, and JavaTest agent.

prior status A JavaTest filter used to restrict the set of tests in a test run based on the last test result information stored in the test result files (`.jtr`).

R

reference implementation (RI) The prototype or proof of concept implementation of a Java technology specification. All new or revised specifications must include an RI. A specification RI must pass all of the TCK tests for that specification.

S

signature file A text representation of the set of public features provided by an API that is part of a finished TCK. It is used as a signature reference during the TCK signature test for comparison to the technology implementation under test.

signature test A TCK signature test for a Java technology implementation checks that all the necessary API members are present and that there are no extra members which illegally extend the API. It compares the API being tested with a reference API and confirms if the API being tested and the reference API are mutually binary compatible.

specification A plan or blueprint for structuring and delivering information.

See Java technology specification.

standard values A configuration value used by the JavaTest harness to determine which tests in the test suite to run and how to run them. The user can change standard values using either the Question mode or Quick Set mode in the configuration editor.

system configuration Refers to the combination of operating system platform, Java programming language, and JavaTest harness tools and settings.

T

tag test description A test description that is embedded in the Java language source file of each test.

Technology Compatibility Kit (TCK) The suite of tests, tools, and documentation that allows an implementor of a Java technology specification to determine if the implementation is compliant with the specification.

technology implementation Any binary representation of the form and function defined by a Java technology specification.

technology specification See Java technology specification.

test agent A test agent is a Java application that receives tests from the test harness, runs them on the implementation being tested, and reports the results back to the test harness. Test agents are normally only used when the TCK and implementation being tested are running on different platforms. See also test agent, passive agent, and JavaTest agent.

test The source code and any accompanying information that exercise a particular feature, or part of a feature, of a See also technology implementation to make sure that the feature complies with the Java technology specification's compatibility rules. A single test may contain multiple test cases. Accompanying information may include test documentation, auxiliary data files, or other resources used by the source code. Tests correspond to assertions of the specification.

test cases A small test that is run as part of a set of similar tests. Test cases are implemented using the JavaTest MultiTest library class. A test case tests a specification assertion, or a particular feature, or part of a feature, of an assertion.

test command A class that knows how to execute test classes in different environments. Test commands are used by the test script to execute tests.

test command template A generalized specification of a test command in a test environment. The test command is specified in the test environment using variables so that it can execute any test in the test suite regardless of its arguments.

test description Machine readable information that describes a test to the test harness so that it can correctly process and run the related test. The actual form and type of test description depends on the attributes of the test suite. A test description exists for every test in the test suite and is read by the test finder. When using the JavaTest harness, the test description is a set of test-suite-specific name/values pairs in either HTML tables or Javadoc-style tags.

test environment A test environment consists of one or more test command template that the test script uses to execute tests and set of name/value pairs that define test description entries or other values required to run the tests.

test execution model The steps involved in executing the tests in a test suite. The test execution model is implemented by the test script.

test finder When using the JavaTest harness, a nominated class, or set of classes, that read, verify, and process the files that contain test description in a test suite. All test descriptions that are located or found are handed off to the JavaTest harness for further processing.

test harness The applications and tools that are used for test execution and test suite management. The JavaTest harness is an example of a test harness.

test script A Java class whose job it is to interpret the test description values, run the tests, and then report the results back to the JavaTest harness. The test script must understand how to interpret the test description information returned to it by the test finder.

test specification A human-readable description, in logical terms, of what a test does and the expected results. Test descriptions are written for test users who need to know in specific detail what a test does. The common practice is to write the test specification in HTML format and store it in test suite's test directory tree.

test suite A collection of tests, used in conjunction with the JavaTest harness to verify compliance of the licensee's implementation of Java technology specification. All TCKs contain one or more test suites.

W

work directory A directory associated with a specific test suite and used by the JavaTest harness to store files containing information about the test suite and its tests.

Index

D

E

F

G

H

I

J

K

L

M

U

V

W

X