Java™ Device Test Suite Developer's Guide

Version 2.4
Java ME Platform

Sun Microsystems, Inc.
www.sun.com

May 2009

Submit comments about this document at: http://java.sun.com/docs/forms/sendusmail.html

Contents

Figures

Tables

Code Examples

Preface

The *Java™ Device Test Suite Developer's Guide* describes how to develop test packs that can integrate with the Java Device Test Suite.

Before You Read This Book

To fully use the information in this document, you must have moderate knowledge of the online help and topics discussed in these books:

- *Java Device Test Suite Tester's Guide*
- *Java Device Test Suite Administration Guide*

You must also have thorough knowledge of the specification and API of the technology such as MIDP, CLDC, MMAPI, OTA, or WMA for which you are writing tests.

In addition, a familiarity with the Ant build utility (see `http://ant.apache.org/`) can be useful.

How This Book Is Organized

Part 1, "Getting Started" gives an overview of concepts and provides hands-on experience building test packs included with this distribution.

Part 2, "Essentials" gets deeper into essential topics of test pack development.

Part 3, "Advanced Topics" covers more advanced topics of test pack development, concepts and procedures that are used in special cases.

Part 4, "Legacy" covers topics and procedures that are maintained for backwards compatibility with previous versions of Java Device Test Suite.

Part 5, "Appendices" contains supplementary information about this release.

Using Operating System Commands

This document does not contain information on basic Solaris™ operating system or Windows commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris operating system documentation, which is at `http://docs.sun.com`

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. `% You have mail.`
`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 *Developer's Guide*. These are called *test packs*. You *must* be an administrator to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Terminology Conventions

The Java Device Test Suite 2.0 introduces the term *test pack* to describe a container of test packages. Previously, this container was known as a *test suite*. The term *suite* still appears in some directory names, file names, and property names. This was done for backward compatibility.

Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

The Java Device Test Suite documentation is divided among manuals and online help. The online help is also provided in printable (PDF) format. For subjects that relate to graphical user interface menus, displays, and controls, consult the online help first. The manuals cover only subjects that are not related to graphical user interface features.

Application	Title
Test Development	This guide
Test Execution	Online help *Java Device Test Suite Tester's Guide* *Java Device Test Suite Test Notes*
Administration	Online help (administrator harness edition) *Java Device Test Suite Administration Guide*

Accessing Sun Documentation Online

The Sun Developer Network program web site enables you to access Java platform technical documentation on the web at http://java.sun.com/reference/docs/index.html.

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Send us your comments at http://java.sun.com/docs/forms/sendusmail.html.

PART I Getting Started

This part will help you quickly get started as a test developer for the Java Device Test Suite. Concepts such as the major kinds of test are introduced. You will be guided to obtain necessary software tools and how to set up your workstation environment. You will given the simple instructions to build some basic test packs that are included with the distribution. And you will be guided to create your first test pack (one of the major kinds of test.)

Chapter 1 describes test pack types and introduces their development.

Chapter 2 describes how to set up the Developer's Kit.

Chapter 3 describes a simple test pack's files and shows you how to build, test, and package it.

CHAPTER 1

Overview

This chapter introduces test pack development, including descriptions of the kinds of test packs and tests you can write. The *Java Device Test Suite Administration Guide* describes how to install a test pack that you have written.

The chapter covers the following subjects:

- Test Packs
- Test Pack Types
  - Runtime Tests
    - Automated Tests
    - Interactive Tests
    - Network Tests
    - Distributed Tests
  - Benchmark Tests
  - Over-the-Air Tests

Read this chapter for an introduction to essential concepts and terms. If you prefer to begin with interactive examples, start by reading Chapter 2and Chapter 3 and then return to this chapter.

Test Packs

The *test pack* is the fundamental unit of test development. It is a collection of tests that evaluate a coherent set of device capabilities, that is, those which have a close functional relationship. Often these tests are derived from a single specification. For example, the MMAPI test pack that Sun supplies with the Java Device Test Suite

contains tests of facilities defined in the Mobile Media API specification (JSR 135). A test pack is also the unit that you can install with the administrator edition of the harness or by a designated Java Device Test Suite administrator.

Independent of its functional orientation, every test pack is one of three types:

- **Runtime –** The tests in a runtime test pack verify the quality of services provided by the test device. A runtime test has a Boolean result: It either passes or it fails.
- **Benchmark –** The tests in a benchmark test pack measure the performance of features that affect the device's user experience. These include the speed with which critical operations execute and the load they place on the device. A benchmark test returns measurements for the user (tester) to evaluate. You must segregate benchmark tests into separate test packs, because benchmark tests produce different results than runtime and OTA tests.
- **Over-the-air (OTA) Provisioning –** OTA tests exercise a device's ability to download and install applications, and to correctly authenticate and authorize such applications. Developers specify parameters for this code and develop MIDlets (applications) to associate with each test. Segregate OTA tests from runtime and benchmark tests, because OTA tests interact differently with the harness.

Because runtime, OTA, and benchmark tests must be separated into different test packs, you must develop three test packs for a given specification if you want to test performance, provisioning, and on-device functions.

Specify a test pack's type in the test pack's `testsuite.info` file (see Chapter 6). The test pack type tells the harness how to run the tests in a test pack.

Test Pack Types

Each type of test pack can contain tests of one or more test types. The pack type determines the possible test types. Tests of different types implement or extend different APIs and behave differently when they run.

The descriptions in this section are for test devices whose Java technology-based implementations can communicate with the harness. Users run tests differently on so-called offline devices, which do not have working Java technology-based communications methods. See the harness online help for more information on offline device testing.

Runtime Tests

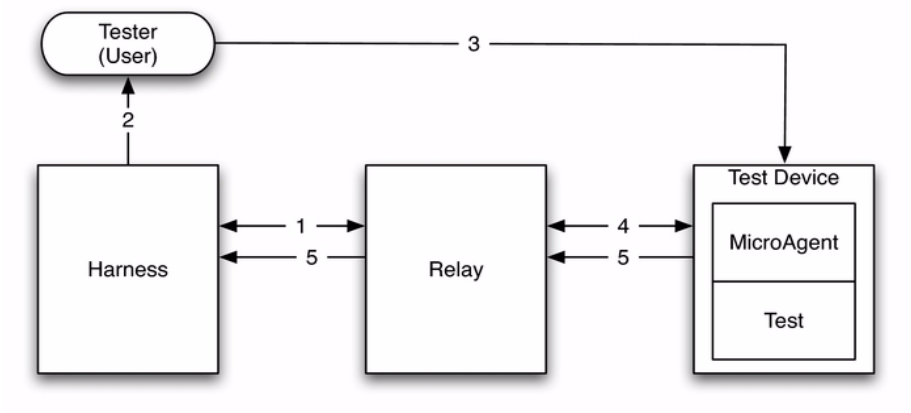
The Java Device Test Suite supports these runtime test types:

- *Automated* tests run entirely on the test device and require no user intervention through the harness. However, some automated tests can require a user response through the test device, for example, through device-specific security prompts.
- *Interactive* tests run entirely on the test device but require user interaction to manipulate the device, observe its response, and decide whether a device has passed or failed the test. Interactive tests include online instructions.
- *Network* tests consist of two parts. One part is an automated test that runs on the test device. The other part runs on the Relay host and acts as a server. The host part receives messages sent from the device-resident part, and can also send messages to the device-resident part.
- *Push* tests test or use the push registry defined by the MIDP 2.0 specification. Some push tests are a special kind of network test that includes a MIDlet that is awakened by a clock or incoming connection.
- *Distributed* tests consist of three parts. One part is a test that runs on the test device. The second part runs as a server on the Relay host. It communicates with the test device part. The third part is the partner MIDlet, which runs on a second device. The partner MIDlet sends and receives messages from the test device.

Automated Tests

Automated tests run without user intervention and automatically return passed or failed status. FIGURE 1-1 shows how the user, the test device, and Java Device Test Suite interact to run an automated test.

FIGURE 1-1 Runtime Automated Test Flow

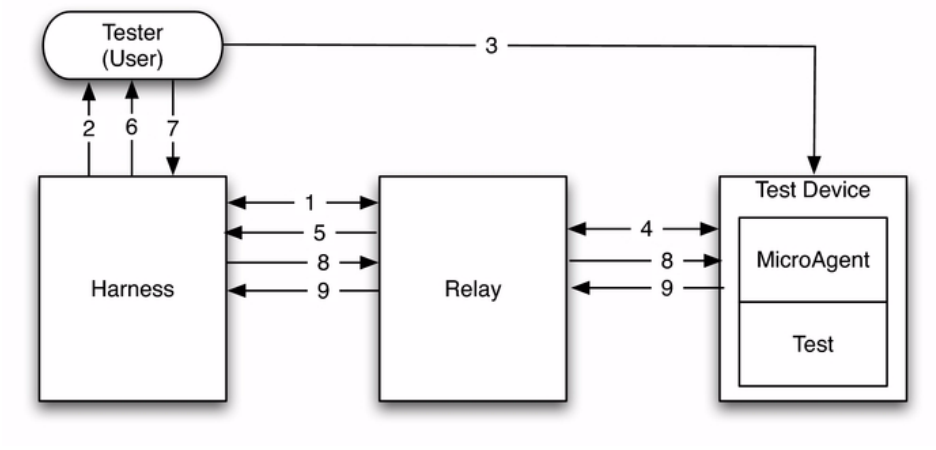


1. After the tester selects tests and presses the harness's Start button, the harness creates and sends the test bundle to the Relay. The Relay provides the harness with the test bundle's URL address. A *test bundle* contains a Java Device Test Suite MicroAgent and one or more tests. The MicroAgent provides infrastructure for the tests and communicates with the Relay. The Relay is an intermediary between the harness and the test device that simplifies communication across firewalls. The Relay also hosts the test server manager (TSM) and servers used by some tests.
2. The harness displays the URL to the tester.
3. The tester discloses the test bundle URL to the test device and initiates the test bundle download process on the device.
4. The test device requests the test bundle, downloads it from the Relay, and launches the MicroAgent. (If the device supports the autotest protocol, it initiates the bundle download automatically.) There are several ways to download bundles, depending on the capabilities of the test device. The online help describes test downloading in detail. The MicroAgent starts the test.
5. The test runs and returns a result of passed or failed to the MicroAgent. The MicroAgent sends the result to the Relay, which sends it to the harness.

Interactive Tests

Interactive tests require the user to operate test device features and to decide if the implementation passes or fails the test. FIGURE 1-2 shows how the user's role expands for interactive tests.

FIGURE 1-2 Runtime Interactive Test Flow



1. As for an automated test, after the tester selects tests and presses the harness's Start button, the harness creates and sends the test bundle to the Relay. The Relay provides the harness with the test bundle's URL address.

2. The harness displays the URL to the tester.

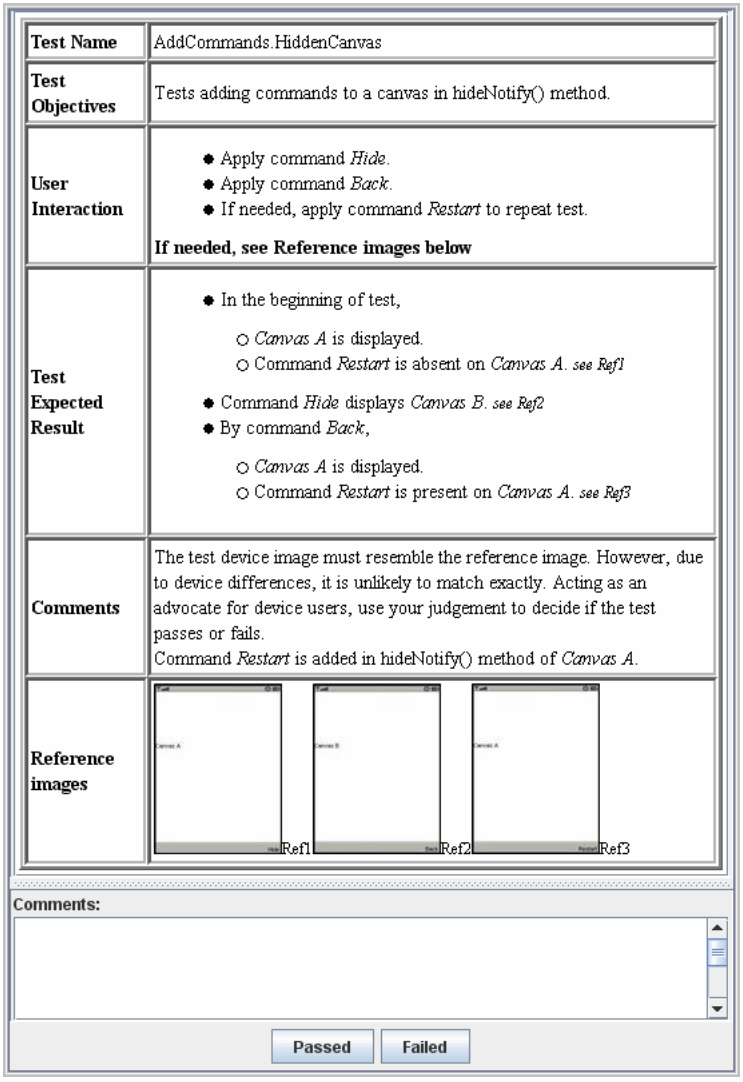
3. The tester discloses the location of the test bundle (URL) to the test device and initiates the test bundle download process on the device.

4. The device downloads the bundle and launches the MicroAgent. (If the device supports the autotest protocol, it initiates the bundle download automatically.) The MicroAgent starts the test in a new thread, tells the Relay to display the test instructions, and begins polling the Relay for the test result.

5. The Relay tells the harness to display the test instructions.

6. The harness displays test instructions and evaluation buttons for the test. The test developer has created these instructions in an HTML file. FIGURE 1-3 shows an example of what the tester sees. The tester follows the instructions, manipulating the device and observing its response.

FIGURE 1-3 Example Interactive Test Evaluation Dialog Box

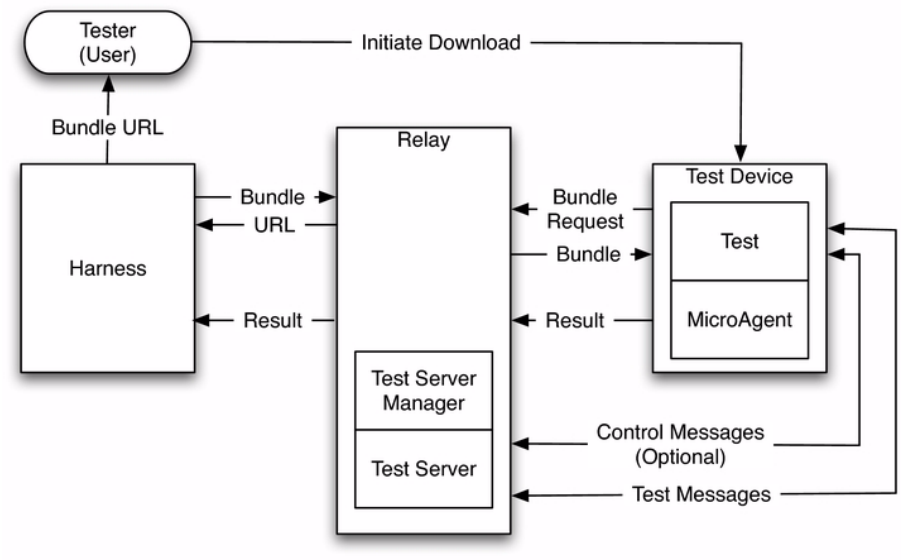


7. The tester clicks the Passed or Failed button, which sends a message to the harness.
8. The harness forwards the result to the Relay, which forwards it to the MicroAgent the next time the MicroAgent polls.
9. As in an automated test, the MicroAgent returns the result to the Relay, which returns it to the harness.

Network Tests

From a user's perspective, a network test is identical to a runtime automated test. From a developer's perspective, a network test has a more complex implementation than an ordinary automated test. A network test has two parts. One part, called the *client*, is a runtime automated test. The other part, called the *server*, runs on the Relay. It emulates a message sender, a message receiver, or both. Network tests by definition can send and receive messages, such as HTTP messages or push notifications.

FIGURE 1-4 Network Test Flow

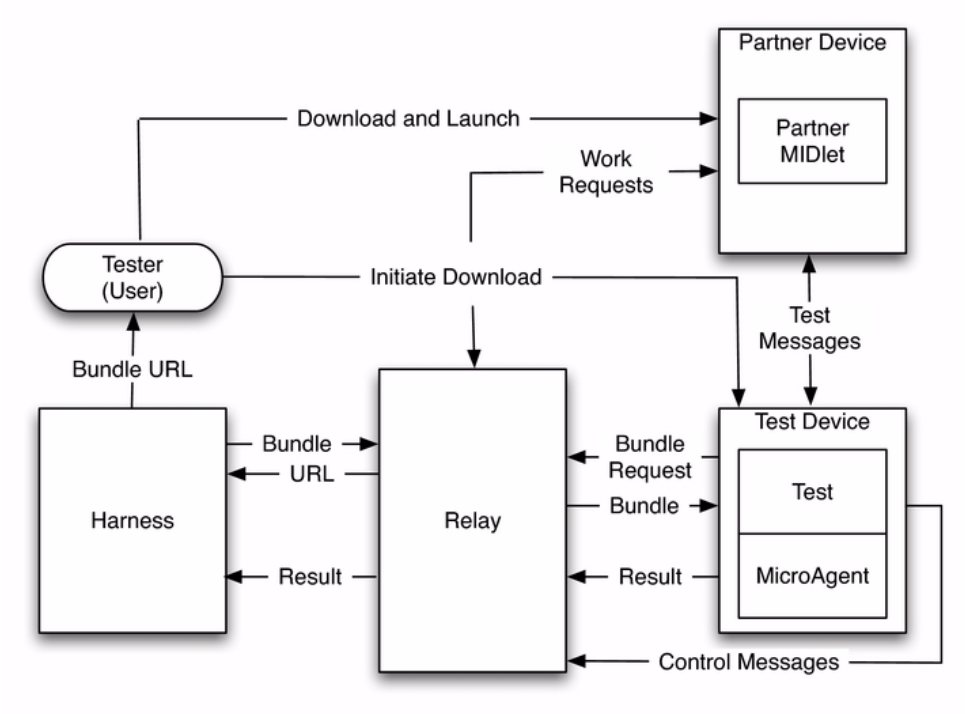


The Java Device Test Suite Relay provides infrastructure for the server side of the network tests. You write only the server code that is specific to your tests. The two parts can optionally communicate through a *control channel* you supply and typically implement with HTTP. For example, a client uses the control channel to tell the server what kind of message to send or what kind of message to expect from the client. Without a control channel, the server part does just one thing (for example, invariably sends a particular message or expects to receive a particular message).

Distributed Tests

A distributed test (see FIGURE 1-5) tests the interaction between different devices. Multiple devices are necessary because the method of communication, such as Bluetooth or SMS, cannot be emulated by a test server running in the Relay.

FIGURE 1-5 Distributed Test Flow



A distributed test has one developer-written part that interacts with two other parts provided by Sun. The developer part (the test itself) is a runtime automated test that runs on the test device. The test interacts with the Relay by sending control messages that indirectly influence the partner MIDlet, which is provided by Sun. The partner MIDlet runs on a partner device (another phone) and communicates with the Relay and the test. A few tests use more than one partner device, for example, to test sending an MMS message to multiple recipients.

The parts operate as follows.

1. The tester downloads the partner MIDlet into the partner device, launches it, and configures it.
2. The partner MIDlet begins polling the Relay for work to do.
3. The test is downloaded to the test device in the usual way and begins to run.
4. The test sends a control message to the Relay which asks the partner MIDlet to perform some work, such as sending a message or waiting for a message.
5. When it receives the next polling request from the partner MIDlet, the Relay passes the test's request to the partner MIDlet.

6. The test and the partner MIDlet communicate to execute the work of the test.
7. The test returns a passed or failed result to the MicroAgent, which in turn returns the result to the Relay, which passes it to the harness, which displays and saves the result.

Benchmark Tests

From a user's point of view, benchmark tests behave like automated runtime tests (see "Automated Tests" on page 3). However, for an implementation to pass or fail a benchmark test, the test must have a *threshold* against which the user evaluates the test run. Users create the threshold by running the test against a reference device and recording the result in a threshold file. The *Java Device Test Suite Tester's Guide* has instructions for creating threshold data, and the *Java Device Test Suite Administration Guide* describes how to create a threshold file from that data. A benchmark test returns one or more measurements to the harness. The harness displays the measurements. If the test has a threshold, the harness also displays the test's pass or fail status.

Over-the-Air Tests

Over-the-air (OTA) provisioning tests verify that a test device's Application Management System (AMS) operates correctly and interacts correctly with the provisioning emulator on the Relay. The provisioning emulator supplies applications over the air to wireless devices.

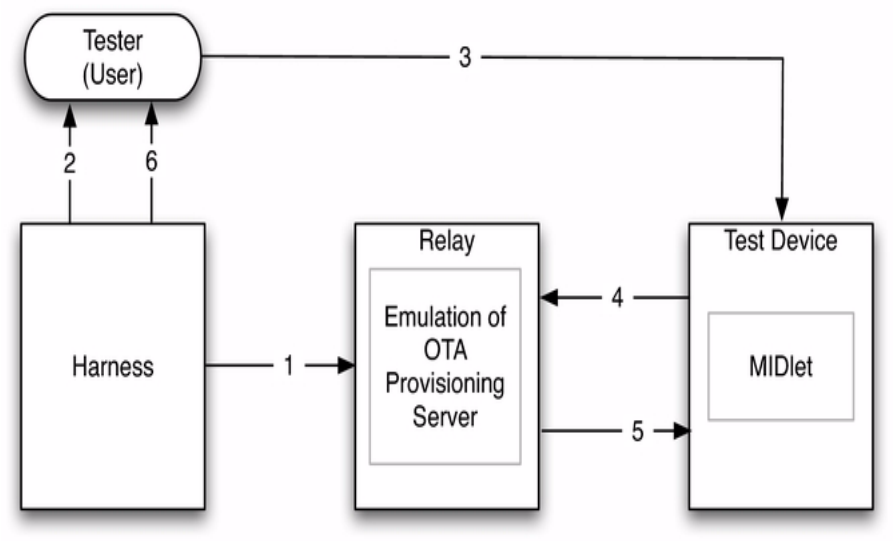
An OTA provisioning test has two parts. One part is the OTA test management Relay's infrastructure. The other part is a MIDlet (application) that you write and the user downloads from the provisioning server, located within the Relay, to the test device. In fact, a test might have multiple MIDlets, but for simplicity only one is considered here. The MIDlet can be merely a placeholder for an application that a test device obtains from a real provisioning server. Or the MIDlet can perform operations on the test device. For example, a MIDlet can verify that the test device correctly grants it permission to use certain APIs.

Java Device Test Suite has two OTA provisioning test types: interactive and semi-automated tests.

OTA Interactive Tests

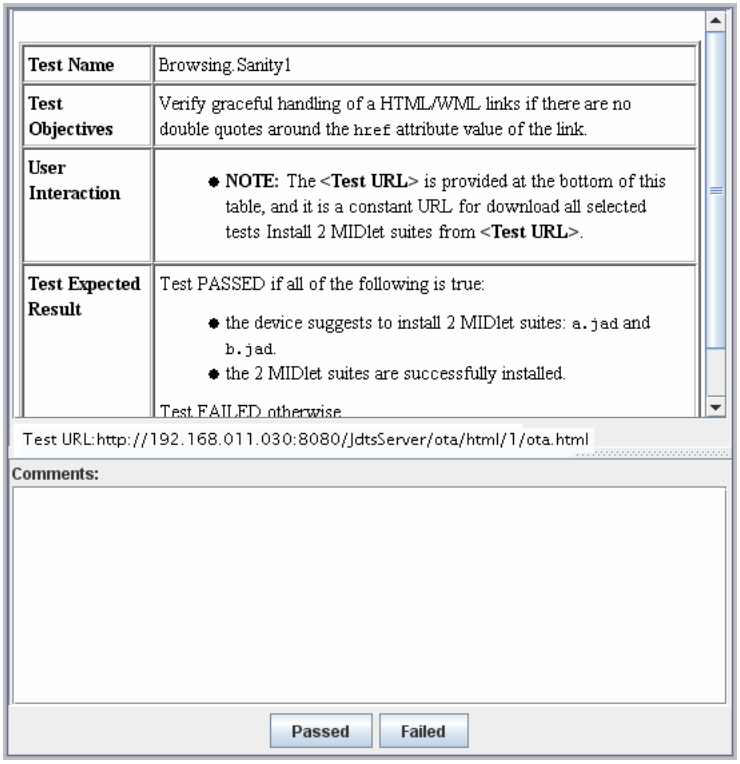
From a user's perspective, an interactive OTA provisioning test is similar to a runtime interactive test (see "Interactive Tests" on page 4). An interactive OTA provisioning test might test the user-visible aspects of provisioning, for example, downloading, installing, and removing a MIDlet. Or, after being installed and launched by the user, the MIDlet might perform more elaborate tests. FIGURE 1-6 shows the interactions.

FIGURE 1-6 OTA Interactive Test Flow



1. The server manages the execution of the test according to developer-specified properties. The user does not download OTA tests but does download MIDlets associated with OTA tests. The harness coordinates with the server much as if the server were the MicroAgent on a test device. After the user selects a test and clicks the harness's Start button, an interactive window displays that specifies the URL; the tester then commands the device to download an application from the specified URL. The server handles this application request.
2. The harness displays test instructions and evaluation buttons to the tester. These are similar to runtime interactive test instructions (FIGURE 1-7 shows an example).

FIGURE 1-7 Sample OTA Provisioning Interactive Test Evaluation Dialog Box



3. Following the instructions, the tester directs the test device to download the MIDlet whose URL is provided in the instructions.
4. The test device sends a request for the MIDlet to the Relay.
5. The Relay usually returns a Java Application Descriptor (JAD) file to the test device. The test device uses the JAD file to download and install the MIDlet. The Relay can also return a Java Archive (JAR) file, depending on the nature of the test. The JAR file installs in the test device immediately.
6. The tester launches the MIDlet. If the MIDlet displays its own instructions on the test device, the user follows them and evaluates the test device's response. The user presses the Passed or Failed button in the test instructions accordingly. The results are then displayed in the harness and saved in the Relay.

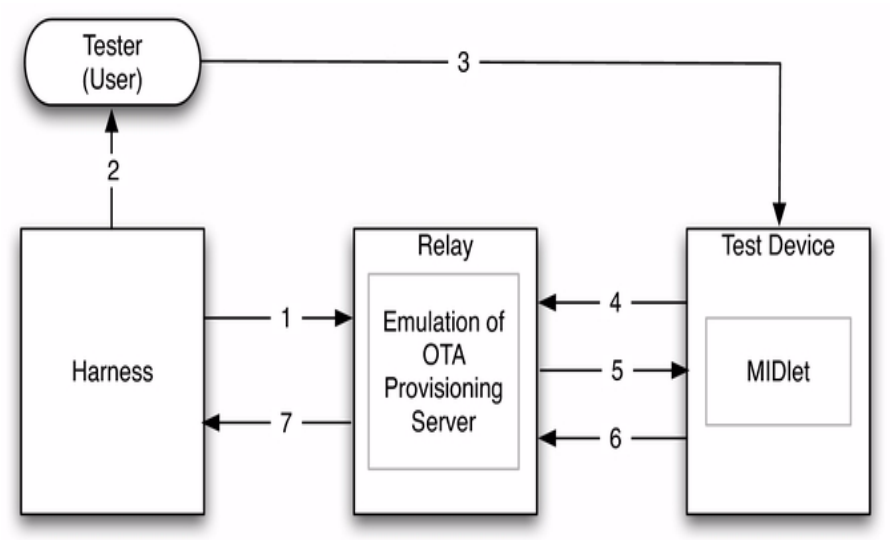
OTA Semi-Automated Tests

Semi-automated OTA tests are similar to interactive OTA tests with the following differences:

- The test device's AMS sends a notification (defined by the OTA provisioning specification) to the server to notify it of an action the device has taken.
- The server examines the notification and determines whether the implementation passed or failed the test.

FIGURE 1-8 shows how a user, a test device, and the Java Device Test Suite components interact.

FIGURE 1-8 OTA Semi-Automated Test Flow



1. After the tester selects a semi-automated OTA provisioning test and presses the harness's Start button, the harness directs the server to start the test.
2. The harness displays the test's test instructions. These are identical to interactive OTA test instructions (see FIGURE 1-7) except that they do not have Passed and Failed buttons. They have a Skip button that the tester can use if a test hangs.
3. The tester directs the test device to download, install, and launch the MIDlet whose URL is given in the instructions.
4. Responding to the user's manipulations, the test device send a request to the Relay for a MIDlet.
5. The Relay returns a JAD file that describes the MIDlet. The test device uses the JAD file to download and install the MIDlet.
6. The test device sends a notification to the server as required by the OTA provisioning specification.

7. The server compares the notification to an expected value (specified in a property) and returns a passed or failed result to the harness.

CHAPTER 2

Setting Up the Developer's Kit

This chapter describes the steps you need to take before writing a test pack. These steps verify that your environment is set up so that tests can be built and tested. It has these sections:

Acquire and Install the Prerequisite Software

To develop a test pack, you must first acquire and install the following software:

- Java Platform, Standard Edition (J2SE™ platform) JDK software. Note the following:
  - The officially supported version is 1.6.0_3 (also known as JDK 6 Update 3). You can use a later version. Sun recommends using the latest JDK software.
  - Download the JDK software from the Sun web site at `http://java.sun.com/javase/downloads/index.jsp`.
  - Add the path to you installed JDK version to the *PATH* environment variable, for example: `C:\Program Files\jdk1.6_03\`.
  - Verify the installation and version with this command:

    ```
    > java -version
    ```

  - No matter which JDK version you use, the tests you build are compatible with Java Device Test Suite harnesses whether they use 1.5 or 1.6 versions of the Java runtime environment.

- The Java Device Test Suite administrator bundle. The *Java Device Test Suite Administration Guide* describes how to obtain, install, and verify the software. Choose the Full Installation option. In this guide, *installDir* refers to directory containing your private copy of the Java Device Test Suite Central Installation. This directory contains the `admin/` directory, `ReleaseNotes.html`, and other files and directories. Its default name is similar to .../`JDTS-CI/`.

Note – To write and test a test pack, you must have a private copy of the full administrator installation, including the Central Installation and the Sun Java System Application Server. To prevent interference with testers and administrators, do not use the production installation for test development. To avoid confusion with the Central Installation that testers use to run tests, this guide refers to your private Central Installation as the Developer Installation.

- Ant 1.7.1 or later. Ant is a build utility written in the Java programming language. For information, see `http://ant.apache.org`.
  - Add the path to the directory containing the installed Ant launch scripts to the *PATH* environment variable, for example, `C:\Program Files\apache-ant-1.7.1\bin\`.
  - Verify the installation and version with this command:

    ```
    > ant -version
    ```

- A MIDP implementation and preverifier for your development platform. Building a test pack requires use of the platform-specific MIDP preverifier. If you are developing tests for a MIDP 1.0 device, you need a MIDP 1.0 implementation. If you are developing tests for a MIDP 2.0 device, you need a MIDP 2.0 implementation. The MIDP implementation and preverifier is available in the Sun Java Wireless Toolkit for CLDC.

  Note these operating system-specific requirements:

  - **Windows platform developers –** Download and install version 2.5 of the Sun Java Wireless Toolkit for CLDC from `http://java.sun.com/products/sjwtoolkit/`. This guide refers to the installation directory as *WTKInstallDir*.
  - **Solaris platform developers –** Download and install version 2.1_01 of the J2ME Wireless Toolkit from `http://java.sun.com/products/sjwtoolkit/download-2_1.html`. This guide refers to the installation directory as *WTKInstallDir*.

    Version 2.1_01 of the Wireless Toolkit cannot be installed with JDK version 1.6.0_03, which is recommended for the Java Device Test Suite. Specify a JDK version in the 1.4 or 1.5 series when you install the WTK on the Solaris platform.

To obtain the latest preverifier, also download and install *on a Windows platform* version 2.5 of the Sun Java Wireless Toolkit for CLDC from `http://java.sun.com/products/sjwtoolkit/`. This guide refers to the installation directory as *winWTKInstallDir*. Copy *winWTKInstallDir*`/lib/jsr082.jar` to *WTKInstallDir*`/lib/`.

Solaris platform developers can use JAR files created on the Windows platform because they are platform independent.

Unzip and Configure the Developer's Kit

The developer's kit is packed into *installDir*`/admin/shared/devkit.zip`. Prepare it for use by following these steps:

1. **Create a directory in which you want to develop your test pack.**

   This guide refers to the directory you create as *devKitHome*.

2. **Unzip** *installDir*`/admin/shared/devkit.zip` **into** *devKitHome*.

3. **Open** *devKitHome*`/tests/common/build/environment.properties` in a text editor.

4. **Verify that** `jdts.home` **is set to** *installDir***. Change the value if necessary.**

5. **Solaris operating system users, skip to** Step 7**.**

6. **Windows operating system users:**

   a. **Set** `client.platform.home` **to the directory containing the Wireless Toolkit for CLDC, for example,** `c:/WTK25`**.**

      Note the forward slash path separator.

   b. **Skip to** Step 8**.**

7. **Solaris operating system users:**

   a. **Set** *client.platform.home* **to** *WTKInstallDir,* **the directory containing the Wireless Toolkit, for example,** `/home/lee/bin/WTK21`**.**

   b. **Remove** `.exe` **from the end of the following line:**

      ```
      preverify.exec=${client.platform.home}/bin/preverify.exe
      ```

8. **Save** `environment.properties`**.**

Build and Install the Sample Test Packs

Building and installing the sample test packs verifies that your developer's kit installation is ready for test pack development. Follow these steps:

1. **If it is not running, start the Relay by starting the Sun Java System Application Server.**

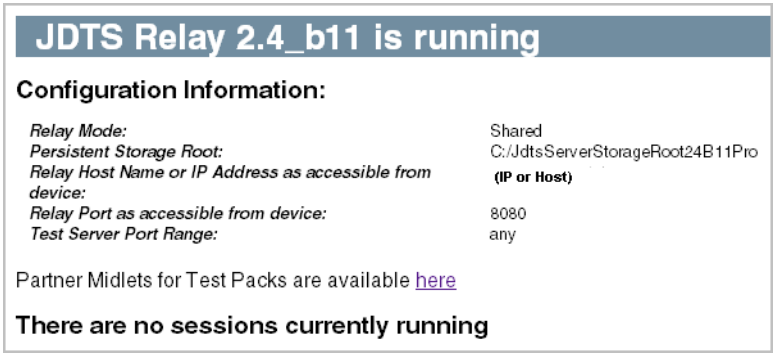
   The *Java Device Test Suite Administration Guide* describes how to start the Relay.

   You can verify that the Relay is running by typing a URL like this into a web browser: `http://`*appServerHost*`:`*appServerPort*`/`*appContext*`/`

   For example: `http://localhost:8080/JdtsServer/`. You specified the values of *appServerHost*, *appServerPort*, and *appContext* when you installed your developer installation.

   If the Relay is running, it displays a page similar to FIGURE 2-1. The version number and other details might be different.

FIGURE 2-1 Relay is Running Page



2. **In a terminal or command window, change to** *devKitHome*`/tests/`**.**

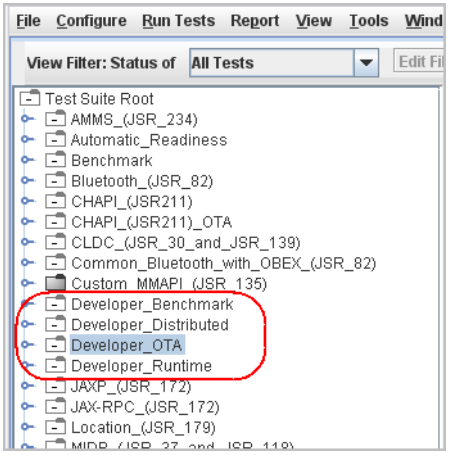
3. **Run this command:**

   ```
   ant -f demo.xml
   ```

   The command can take several minutes to run. It produces hundreds of lines of console output.

4. **When the command completes, launch the graphical administrator harness and verify that test tree shows the Developer test packs shown in** FIGURE 2-2**.**

FIGURE 2-2 Administrator Harness with Developer Test Packs Installed



CHAPTER 3

Introducing the Developer's Kit

Although writing tests is your responsibility, the Java Device Test Suite Developer's Kit contains components you can modify, or, in some cases, use as is. This chapter introduces many of those components by examining, slightly modifying, and building the sample runtime test pack included in the Developer's Kit.

To follow this section, begin with a fresh copy of the unzipped Developer's Kit (see "Unzip and Configure the Developer's Kit" on page 17). Do not run *devKitHome*`/tests/demo.xml` in this copy of the Developer's Kit.

This chapter has these sections:

Developer's Kit Structure

This section introduces the directories and files in the Developer's Kit, focusing on those that implement the sample runtime test pack. The sample benchmark test pack has a very similar structure. The distributed and OTA samples have additional structures to support the more complicated execution models of their tests.

devKitHome

TABLE 3-1 shows the contents of the Developer's Kit top level directory.

TABLE 3-1 *devKitHome* Directory Contents

devKitHome	
`tests/`	Build scripts and sample test packs
`docs/`	Javadoc™ tool documentation for test APIs
`tpim.sh` and `tpim.cmd`	Scripts for faster test pack installation, see Chapter 19
`version.id`	Developer's Kit version identifier. Do not change this file.

`tests/`

This directory contains three sample test packs, a set of common resources used by all test packs, and the `demo.xml` file you used to build and install the sample test packs in Chapter 2. Do not delete this directory.

TABLE 3-2 *devKitHome*`/tests/` Directory Contents

devKitHome`/tests/`	
`common/`	Common resources, notably build scripts in `build/`
`benchmark/`	Sample benchmark test pack
`runtime/`	Sample runtime test pack
`distributed/`	Sample runtime distributed test pack
`ota/`	Sample over-the-air provisioning test pack

tests/common/

Do not delete this directory or its contents. It holds common resources required to build any kind of test pack.

TABLE 3-3 *devKitHome*/tests/common/ Directory Contents

devKitHome/tests/common/	
src/	Common (test-independent) sources for distributed test packs
lib/	Libraries for build
build/	Build scripts and configuration files common to all test packs. .properties files specify default build options.

tests/runtime/

This directory contains the sample runtime test pack. If you are writing a benchmark or OTA test pack, you can delete this directory.

TABLE 3-4 *devKitHome*/tests/runtime/ Directory Contents

devKitHome/tests/runtime/	
build/	Files that control the build for this test pack. Every test pack must have these files. You can edit these files to override default values specified in build/. build.xml is the main Ant build file.
src/	Source and related files that implement the tests in this test pack. Every test pack must have a directory like this.
src/client/	Source and related files for test components that run on test devices
src/server/	Source and related files for the server side of network tests
src/server2/	Additional server sources (directory is named in testsuite.info TestServerSourcesDir property)

TABLE 3-4 *devKitHome*`/tests/runtime/` Directory Contents *(Continued)*

`testserver_resources/`	Sample server resources for network tests
`testsuite.html`	Top level documentation for this test pack. Every test pack must have a file like this. See "Documenting a Test Pack" on page 45 for more information.
`testsuite.info`	Global test suite properties. Every test pack must have a file like this. See Chapter 6 for details.
`testpack.version.properties`	Test pack version identifier. See Chapter 20 for details.

Editing and Building a Test Pack

Exactly how you create a test pack is subject to variations, such as whether you assemble the files from scratch or begin with a sample. This section introduces one way to begin with the runtime sample test pack.

Note – A real test pack differs substantially from a sample or this trivial modification. Notably, the directory structure in *devKitHome*`/tests/runtime/src/client/` and `server/` will reflect your company, not `com/sun/`. There are many options you can change to suit your needs and preferences. Consult the sample files for examples.

---

1. **Change to** *devKitHome*`/tests/common/build/`**.**
1. **Either edit** `environment.properties` **as you did in** "Unzip and Configure the Developer's Kit" on page 17 **or replace** `environment.properties` **with the corresponding file from the other Developer's Kit.**
2. **Optionally, delete** *devKitHome*`/tests/distributed/`**,** `benchmark/`**, and** `ota/`**.**

   You do not need these files in a runtime test pack.
3. **Change to** *devKitHome*`/tests/runtime/`**.**
4. **In** `testsuite.info` **change the following lines:**

   a. **Change the value of** `TestPackName` **to** `My Runtime`**.**

   b. **Change the value of** `TestSuiteName` **to** `My Runtime TestSuite`**.**

   c. **Change the value of** `TestSuiteID` **to** `com.sun.jdts.devkit.my.runtime`**.**
5. **In** `testpack.version.properties` **change** `1.4` **to ++1.0.**

6. **Change to** `build/`**.**

7. **In** `build.properties`**, append these lines:**

```
doc.copyright=\
\ Copyright 2007 My Company. All rights reserved. \n\
\ Use is subject to license terms.
```

8. **Run one of these commands, depending on your platform:**

```
> ant clean all install
% ant clean all install
```

   The command displays several hundred messages, ending with `BUILD SUCCESSFUL` and the elapsed time.

9. **Launch or restart your administrator harness.**

   The new test pack My_Runtime appears in the test tree. It is ready for testing.

Files Generated by the Build

TABLE 3-5 summarizes the files that the build generates.

TABLE 3-5 Files Generated by the Build

devKitHome`/tests/runtime/`	
`bin/`	Class and other files that constitute the executable test pack
`resources/`	Test pack properties for configuration and template editors.
`tmp-bin/`	Temporary directory deleted by `ant clean`.
`listings/`	Catalog of the test pack components. Can be viewed in a web browser.
`test-doc/`	Javadoc tool-style test documentation that is displayed when a tester selects a package or test and clicks the Documentation tab

Packaging a Test Pack

When you have tested your tests, you must package them in a zip file so an administrator can install them in the production Central Installation with the Configure > Test Packs command. In the test pack's `build/` directory, use one of the following commands, depending on your platform:

```
> ant pack
% ant pack
```

Give the resulting zip file to an administrator.

PART II Essentials

This part takes you beyond the concepts overview and quick start examples to get you deeper into essential topics of test pack development.

Chapter 4 details the guidelines for writing descriptive comment blocks ahead of each test class and each test case in test pack source files.

Chapter 5 details the writing of online documentation that should be included with the test packs you develop.

Chapter 6 details the `testsuite.info` file, a properties file that you should create at the top of the test pack directory.

Chapter 7 introduces the `build.properties` files, which control optional features of the build.

Chapter 8 describes the services provided by the singleton class `Runner`.

Chapter 9 describes how to create a runtime test pack.

Chapter 10 describes how to create a benchmark test pack.

Chapter 11 describes how to create an over-the-air (OTA) provisioning test pack.

.

CHAPTER 4

Test Class and Case Comment Blocks

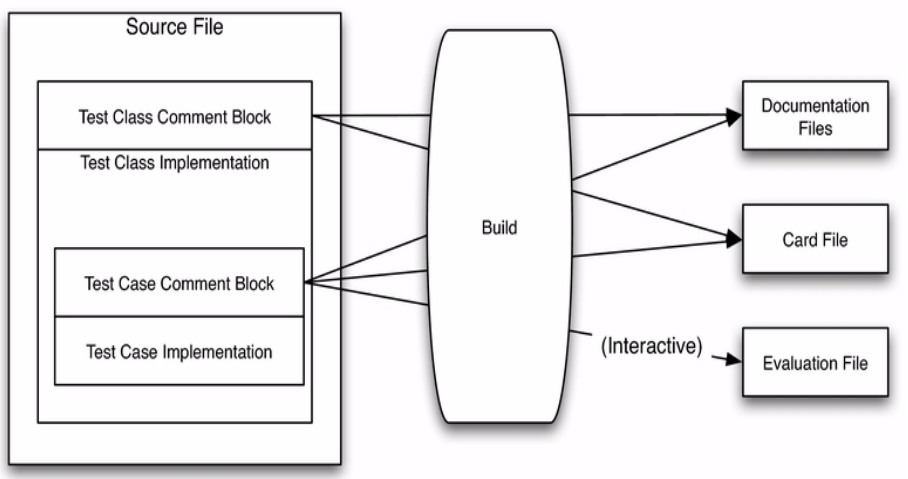
You must precede each test class and each test case with a descriptive comment block. These blocks identify the classes and methods that represent test classes and cases. They also give the build system and the harness information required to build and bundle tests and to display configurable properties to testers. The comment blocks are similar to Javadoc tool “doc comments” described at `http://java.sun.com/j2se/javadoc/writingdoccomments/`

This chapter has the following sections:

Comment Block Overview

When you build a test pack, the build system generates the files shown in FIGURE 4-1 from comment blocks you code for each test class and test case.

FIGURE 4-1 Files Generated from Comment Blocks



The default build generates the following files:

- Documentation files - HTML files created by the Javadoc tool. When a tester clicks the Documentation tab, the graphical harness displays these files. There are additional test pack and package documentation files which you create separately as described in Chapter 5.
- Card file - Test class and case information used by the harness, such as property definitions and a list of related files that test cases require to run. The word "card" has no significance.

  You can alternatively create a card file manually, as described in Chapter 25. You can turn off the autogeneration of card files with a build option.

  You can verify the contents of a card file using the tool described in Chapter 12.
- Evaluation File - For interactive tests only, an HTML file that contains tester instructions for running and evaluating the test. You can alternatively create an evaluation file manually, as described in Chapter 27.

CODE EXAMPLE 4-1 shows a typical test case comment block.

CODE EXAMPLE 4-1 Typical Test Case Block Comment

```
/**
    * Interactive test that asks user to install more than one test MIDlet
   * suite associated with this test case. In order to do that the developer needs
    * to define the <code>JADPath&lt;N&gt;</code> and
    * <code>JARPath&lt;N&gt;</code> properties, where
   * <code>N</code> is decimal integer starting from 1 to infinity (without gaps).
    * For instance, if you need a testcase to install 3 MIDlet suites, use N=1,2,
and 3.
```

CODE EXAMPLE 4-1 Typical Test Case Block Comment *(Continued)*

```
    * Each property contains the pointer to the appropriate test MIDlet.
    *
    * @testcase
    *
    * @precondition none
    *
    * @userInteraction
    * <ul>
    * <li>Install <code>syspropviewer.jad</code> using the
    *    provided URL</li>
    * <li>Select installed MIDlet and launch it</li>
    * <li>Install <code>choicegroup.jad</code> using the
    *    provided URL</li>
    * <li>Select installed MIDlet and launch it.</li>
    * </ul>
    * @postcondition The 2 MIDlet suites are installed
    * @passCriteria User is able to install both of the MIDlet suites and
    *    launch them. The "SysPropertiesViewer" MIDlet should show a list of
    *   system properties describing your device. "ChoiceGroupDemo" application
    *    should display different types of choice group UI.
    *
    * @card.property JADPath1=syspropviewer/syspropviewer.jad
    * @card.property JARPath1=syspropviewer/syspropviewer.jar
    * @card.property JADPath2=choicegroup/choicegroup.jad
    * @card.property JARPath2=choicegroup/choicegroup.jar
    */
void t02InstallMoreThanOneMIDletSuite();
```

FIGURE 4-2 and FIGURE 4-3 show the (rendered) test case documentation and evaluation file generated from the block comment shown in CODE EXAMPLE 4-1.

FIGURE 4-2 Typical Generated Test Case Documentation

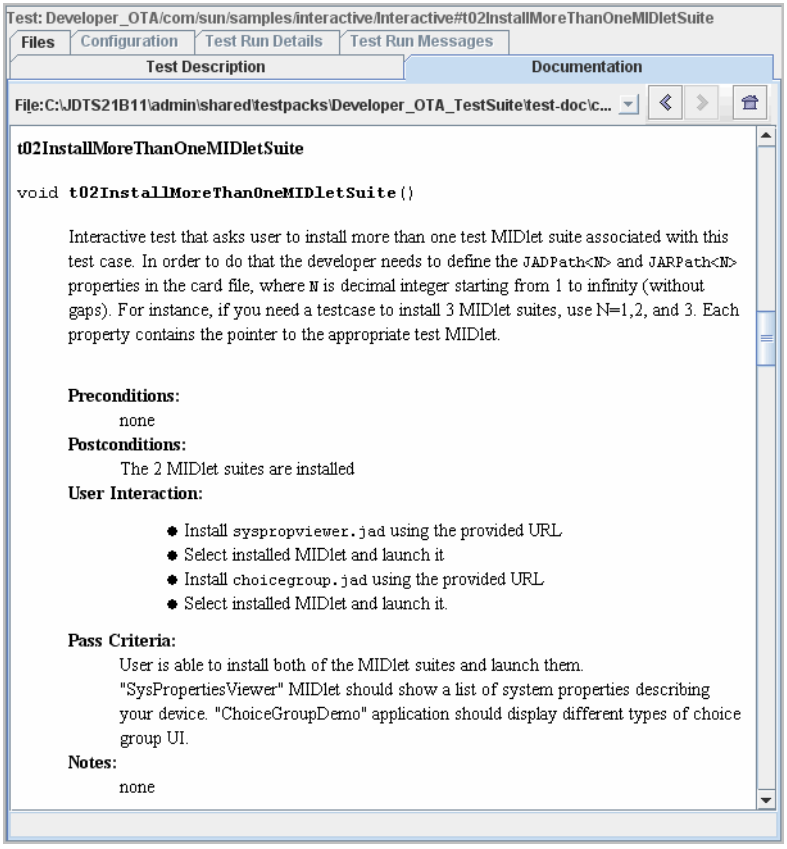
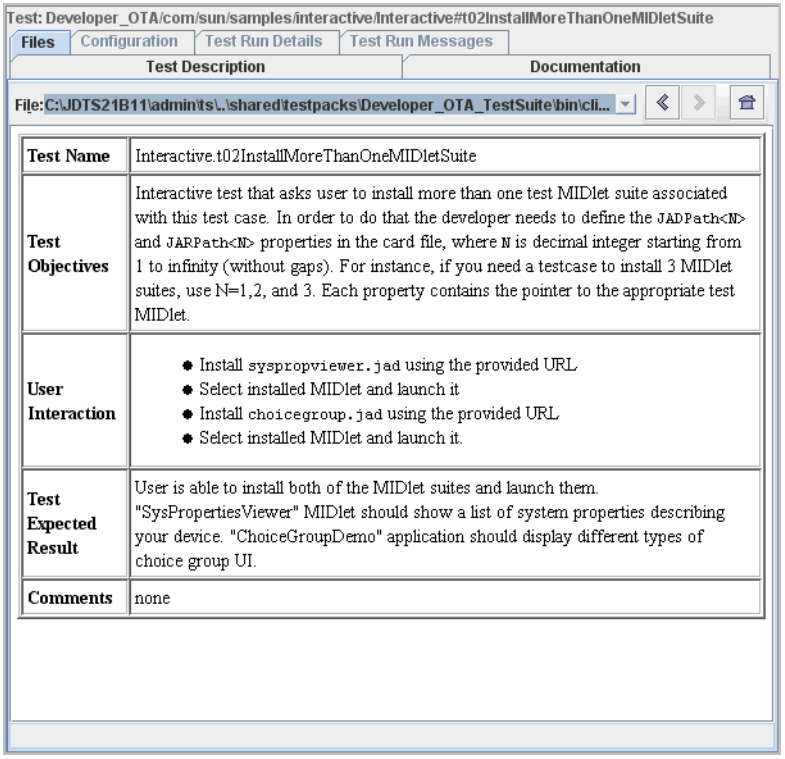


FIGURE 4-3 Typical Generated Interactive Test Evaluation Instructions



Although there is some flexibility in formatting a comment block, observe the following rules:

- Each block begins with a line containing only `/**` and ends with a line containing only `*/`. The class or test case declaration associated with the comment block follows the `*/`.
- Subsections of a comment block are introduced by tags, which have the form `@`*tagName*. The supported tags are defined in TABLE 4-1 and TABLE 4-2. You can also include standard Javadoc tool tags, such as `@since, @throws`. You can use `{@inheritDoc}` in JavaDoc tool tags. Case is significant in tag names. Write `@userInteraction` rather than `@userinteraction`.
- The untagged lines preceding the first tag introduce the class or case. In Javadoc tool terms, these lines are the "main description"
- Each block must have an `@testcase` or an `@testclass` tag.
- If the same tag appears in both the test class comment block and a test case comment block, the precedence rules described in Chapter 16 apply.

- All tags except `@card.*`, `@testcase`, and `@testclass` can be followed by lines of simple (version 4.0.1) HTML to create formatted output such as bullet lists. The lines following `@userInteraction` in CODE EXAMPLE 4-1 give an example.
- `@card.*` tags have these rules:
  - They cannot contain HTML.
  - Lines terminated by a `\` (backslash) character continue on the next line. On the continuation line, the `*` (asterisk) character is ignored. For example:

    ```
    * @card.property x=one \
    *two
    ```

    The value of `x` is `one two`.
  - You can use Java programming language properties file escape sequences. For example: `\\` (backslash), `\u`*HHHH* (Unicode character with hexadecimal value *HHHH*). However, do not use `\n` (line break) in the value of a property. The line break character might cause unexpected behavior. For property file details, see `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html#load(java.io.InputStream)`

Test Class Comment Block Tags

TABLE 4-1 describes the Java Device Test Suite tags that can or must appear in a test class comment block. The Instances column specifies how many instances of a tag are permitted.

TABLE 4-1 Test Class Block Comment Entries

Entry	Test Types	Required?	Instances	Description
/** * *htmlLines*	All	Yes	1	*htmlLines* is the class description in the generated test documentation. In Javadoc tool terms, this is the "main description".
* `@testclass`	All	For test classes	1	Identifies next class definition as a test class.
* `@assumption` *htmlLines*	All	No	0, 1	A condition that is assumed to be true, but cannot be changed, and, in some cases, even checked by the tester. A typical assumption is an interpretation of a related specification statement. If an assumption does not appear to be true, the test should be considered not applicable to the device, which typically means that the test fails if it is run. *htmlLines* is the content of the corresponding section of the generated test documentation and applies to all cases in the class.
* `@precondition` *htmlLines*	All	No	0, 1	A condition that must be true just prior to the execution of the test. If the precondition is not met, the test may give a wrong result. The precondition statement is checkable and achievable: There should be a way to change the environment and guarantee the precondition to be fulfilled. This is typically the tester's responsibility to ensure that the test precondition is fulfilled. Typical precondition is a reminder about some necessary configuration settings. *htmlLines* is the content of the corresponding section of the generated test documentation and applies to all cases in the class.

TABLE 4-1 Test Class Block Comment Entries *(Continued)*

Entry	Test Types	Required?	Instances	Description
`* @postcondition` *htmlLines*	All	No	0, 1	A condition that must be true immediately after successful execution of a test case. "Successful" means no error, unexpected exception, or VM_EXIT. *htmlLines* is the content of the corresponding section of generated test documentation and applies to all cases in class.
`* @keyword` *keywordList*	All	No	0, *n*	Keyword(s) that testers can use to filter (subset for execution) this test case or test class. *keywordList* is a space-separated list of keywords. Interactive tests *must* have this entry: `@keyword interactive`. To see the current list of keywords and their definitions, launch the harness and create or open a work directory. Choose Configure > Edit Configuration or Configure > New Configuration. In the Test Selection section of the interview, answer Yes to Specify Keywords? The More Info pane displays the current list of keywords.
`* @notes` *htmlLines*	All	No	0, 1	Notes that provide test class information to the tester. Quoting from a specification is one use of this tag. If a note is not necessary, do not use the tag.
`* @card.property` *value*	All	No	0, *n*	Defines a property for all test cases in the class. See "`@card.property`" on page 40 for details.
`* @card.specialproperty` *value*	All except OTA	No	0, *n*	Adds a line to the JAD file in a test bundle that contains this test. See "`@card.specialproperty`" on page 41 for details.
`* @card.requires` `[`*value*`]` `[*]`	All	No	0, *n*	Ensures that a test bundle that contains this test also contains all files the test needs to run. See "`@card.requires`" on page 41 for details.
`* @card.attribute` *value*	All	No	0, 2	Defines the factors that determine the severity of a test case failure. See "`@card.attribute`" on page 43 for details. Two of these comments are required (at the case or class level) if you want to give testers the ability to select tests and report results by failure severity, as they can with Sun tests. The *Java Device Test Suite Tester's Guide* describes the severity concept.

Test Case Comment Block Tags

TABLE 4-2 describes the Java Device Test Suite tags that can or must appear in a test case comment block. The Instances column specifies how many instances of a tag are permitted.

TABLE 4-2 Test Case Block Comment Entries

Entry	Test Types	Required?	Instances	Description
`/**` `*` *htmlLines*	All	Yes	1	*htmlLines* is the class description in the generated test documentation. For interactive tests, *value* is the content of Test Objectives in the evaluation file. In Javadoc tool terms, this is the "main description".
`* @testcase`	All	For test cases	1	Identifies next method definition as a test case.
`* @assumption` *htmlLines*	All	No	0, 1	A condition that is assumed to be true, but cannot be changed and, in some cases, even checked by the tester. A typical assumption is an interpretation of a related specification statement. If an assumption does not appear to be true, the test should be considered not applicable to the device, which typically means that the test fails if it is run. *htmlLines* is the content of the corresponding section of the generated test documentation.
`* @precondition` *htmlLines*	All	No	0, 1	A condition that must be true just prior to the execution of the test. If the precondition is not met, the test may give a wrong result. The precondition statement is checkable and achievable: there should be a way to change the environment and guarantee the precondition to be fulfilled. It is typically the tester's responsibility to ensure that the test precondition is fulfilled. A typical precondition is a reminder about some necessary configuration settings. *htmlLines* is the content of the corresponding section of the generated test case documentation.

TABLE 4-2 Test Case Block Comment Entries *(Continued)*

Entry	Test Types	Required?	Instances	Description
* `@postcondition` *htmlLines*	All	No	0, 1	A condition that must be true immediately after successful execution of a test case. "Successful" means no error, no unexpected exception, and no VM_EXIT. *htmlLines* is the content of the corresponding section of generated test documentation.
* `@passCriteria` *htmlLines*	Runtime, OTA	For interactive	0, 1	Condition or conditions that must be met for the test to pass. For interactive tests, *htmlLines* is the content of Test Expected Result section of the test evaluation instructions.
* `@userInteraction` *htmlLines*	Runtime and OTA interactive	For interactive	0, 1	Instructions for tester to execute an interactive test. *htmlLines* is the content of the User Interaction section of the test evaluation instructions.
* `@referenceImages` *URLs*	Runtime and OTA interactive	No	0, 1	*URLs* is URL(s) of image(s) displayed in evaluation instructions.
* `@performanceMetric` *value*	Benchmark	For benchmark	0, 1	Use instead of `@passCriteria`. *value* must be one of: `UnitRate`, `SystemLoad`.
* `@keyword` *keywordList*	All	No	0, *n*	Keywords testers can use to filter (subset for execution) this test case or test class. *keywordList* is a space-separated list of keywords. Interactive tests *must* have `@keyword interactive`. To see the current list of keywords and their definitions, launch the harness and create or open a work directory. Choose Configure > Edit Configuration or Configure > New Configuration. In the Test Selection section of the interview, answer Yes to Specify Keywords? The More Info pane displays the current list of keywords.
* `@notes` *htmlLines*	All	No	0, 1	Notes that provide test case information to the tester. For interactive tests, *htmlLines* is the content of the Comments section of the evaluation file. Quoting a specification or describing unusual test behavior ("On some devices, the screen blinks") are typical uses of notes. If a note is not necessary, do not use the tag.

TABLE 4-2 Test Case Block Comment Entries *(Continued)*

Entry	Test Types	Required?	Instances	Description
`* @card.property` *value*	All	No	0, *n*	Defines a test case property. See "`@card.property`" on page 40 for details.
`* @card.requires` [*value*] [*]	All	No	0, *n*	Ensures that a test bundle that contains this test also contains all files the test needs to run. See "`@card.requires`" on page 41 for details.
`* @card.attribute` *value*	All	No	0, 2	Defines the factors that determine the severity of a test case failure. See "`@card.attribute`" on page 43 for details. Two of these comments are required (at the case or class level) if you want to give testers the ability to select tests and report results by failure severity, as they can with Sun tests. The *Java Device Test Suite Tester's Guide* describes the severity concept.

For tags that can be used with both test classes and test cases, use the class tag to specify values that apply to all cases in a class. Use the case tag to specify values that apply to one case only. For example, suppose you specify the following:

```
  * @testclass
* @keyword interactive
* ...
* @testcase
* @keyword onePartnerMIDlet
```

This means that all test cases in the class can be filtered with the `interactive` keyword but only the specified test case can also be filtered with `onePartnerMIDlet`.

Tag Details

Tags with more complex syntax or semantics are described in this section.

@card.property

If a test class or case needs a user-visible property, name the property and specify its default value with the following syntax:

```
* @card.property PropertyName=DefaultValue
```

For example:

```
* @card.property MaxDistance=12
```

Properties that apply to multiple classes can be defined globally at the test pack level. Such properties are defined in the `testsuite.info` file at the top of the test pack directory in the file system. A property defined at the test pack level can be overridden by defining a property of the same name within the test class or case comment blocks. See "Properties and Parameter Expansion" on page 99 for details.

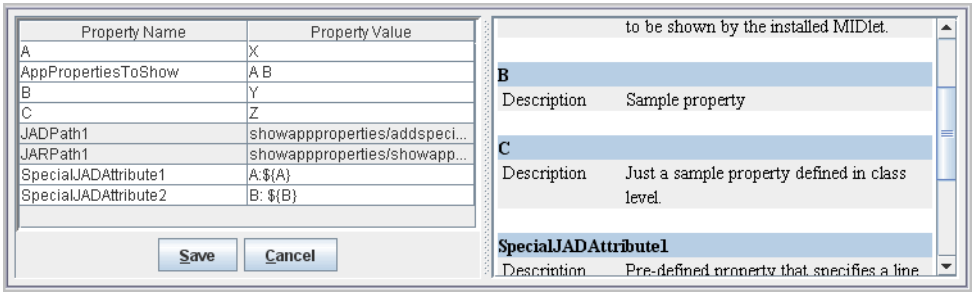
The following test pack property attributes are also supported for test class and case properties.

- `scope`: To hide a property from users, specify the value `hidden`. The `advanced` value is not supported for classes and cases.
- `readonly`: To prevent user modification of the property's default value, specify the value `true`.
- `doc`: Give a short description of the property, for example:

```
* @card.property PixelHeightNoLessThan.doc=minimum number of pixels
allowed for object height
```

FIGURE 4-4 shows an example of `doc` values that appear when a tester right-clicks a test case in the harness test tree and chooses Configure Test. Read-only properties are displayed in gray (for example, JADPath1 in FIGURE 4-4).

FIGURE 4-4 `doc` Properties in Configure Test Window



For the exact syntax of property attributes, refer to "Scope" on page 50, "Read-only Properties" on page 51, and "Online Documentation for a Property" on page 54

@card.specialproperty

Use a special property definition to direct the harness to add a line to the JAD file that it creates for a test bundle containing this test. In effect, a special property definition is a way to pass a parameter to an application management system (AMS), sometimes called a Java application manager. The static parameter applies only to the test bundle associated with the JAD file.

Use the following syntax to specify a line to be added to the JAD file:

```
* @card.specialproperty <jad>.n=LineToAdd
```

n is a number that distinguishes multiple `@card.specialproperty` entries in the same source file. The harness interprets *LineToAdd* in this line as a name:value pair taking the first colon symbol as the separator between the name and the value. The harness copies the name:value pair to the JAD file. The test device AMS interprets the name:value pair when it downloads the JAD file. It is the test developer's responsibility to ensure that name:value pair is meaningful to the AMS and follows the JAD file syntax. The JAD file is defined in the MIDP specification.

For example, to add a line to a JAD file associated with a test bundle containing `MIDlet2:AlarmMidlet`, include a line like this in the test class block comment:

```
  * @card.specialproperty <jad>.1=MIDlet2:AlarmMidlet
```

Parameter expansion, described in "Properties and Parameter Expansion" on page 99, also applies to special property definitions. The samples in *devKitRoot*`/tests/runtime/src/client/com/sun/samples/network/client/push/` illustrate its use.

@card.requires

This tag specifies files that must be in the test bundle for one or more test cases in a class to execute. Examples of required files include media files to play on the test device and helper classes. CODE EXAMPLE 4-2 gives examples for a class and a case:

CODE EXAMPLE 4-2 Example Required File Values

```
 * @testclass
 * ...
 * @card.requires Pic1.jpg
 * @card.requires com/some/apackage/img/Pic2.jpg
 * @card.requires *
 * @card.requires com/some/apackage/Aclass.class *
```

CODE EXAMPLE 4-2 Example Required File Values *(Continued)*

```
 * @testcase
 * ...
 * @card.requires Pic3.jpg
```

Observe the following when writing path names for this tag:

- The path name separator must be a forward slash (`/`) character.
- If the path name contains a separator (for example, `com/some/Foo.class` or `img/Pic.png`), the path is relative to the directory specified in the test pack property `TestClassesDir`. For example, for runtime test packs, this directory is typically `bin/client/verified` (relative to the test pack root).
- If the path name does not contain a separator (as in the `Pic1.jpg` line in CODE EXAMPLE 4-2), the file is in the same directory as the class of the file containing the `@card.requires` tag.
- The special file name of `*` directs the build to automatically find all class files required by the class containing the `card.requires` comment.
- Following the name of a required class, the `*` character means "include this file and find the files required by the required file".
- You can specify class name explicitly as when a class is loaded using `Class#forName(String)`, for example.
- If a test class refers to a constant field defined in another class, the class that defines the constant may not be found by the card file generator. The reason is "inlining" constant values by `javac` compiler. The test case `testCase1` in *devKitHome*`/tests/runtime/src/com/sun/samples/Automated/SampleAutomatedTest.java` demonstrates this effect. See the comments in the classes `IInlinedConstants.java` and `IReferencedConstants.java` for more detail.
- Automatic searching applies to Java programming language `.class` files only. Do not specify an asterisk (`*`) with non-class resources, such as images. For example, the following is an error:

  ```
  * @card.requires Pic1.png *
  ```

The path in an `@card.requires` entry can specify parameter expansion notationas described in "Properties and Parameter Expansion" on page 99. CODE EXAMPLE 4-3 shows an example.

CODE EXAMPLE 4-3 Parameter Expansion in `@card.requires` Example

```
 * @card.property req=a
 * ...
 * @card.requires com/some/samples/automated/${req}.html
```

In this example, changing the value of `req` to `b` will include the file `com/some/samples/automated/b.html` in test bundles.

Note – Properties are not inherited. In particular, required file properties declared in class `X` do not apply to `X`'s superclass or subclasses. Be sure to declare the files that each class requires.

@card.attribute

Each test class and case can have a failure severity indicator, which is computed from two factors, *functionality* and *impact*. TABLE 4-3 shows the three functionality and impact codes and the failure severity values that the Java Device Test Suite computes from them. The *Java Device Test Suite Tester's Guide* gives more information on failure severity.

TABLE 4-3 Severity Calculation from Functionality and Impact

	Impact		
Functionality	**1 - Critical**	**2 - Significant**	**3 - Limited**
1 - Primary	1 - Very High	2 - High	3 - Medium
2 - Secondary	2 - High	3 - Medium	4 - Low
3 - Nonessential	3 - Medium	4 - Low	5 - Very Low

Use the following guidelines to select a functionality value:

- 1 - Primary: The associated Test Objectives are a mandatory requirement of the implementation function tested by the test.
- 2 - Secondary: Indicates a recommended practice of the implementation function in accordance with Test Objectives. There may be valid reasons in particular circumstances to ignore this recommendation of the implementation function.
- 3 - Nonessential: Indicates that the Test Objectives are an optional requirement of the implementation function tested by the test.

Use the following guidelines to select an impact value:

- 1 - Critical: The tested device is effectively unusable as a result of the test failure. The test failure causes critical impact on the operation of the device/implementation. Critical impact should cover the case when implementation does not work. The test failure renders the implementation ineffective.
- 2 - Significant: Device failure causes significant impact. A test failure identifies a serious but predictable and manageable device failure.
- 3 - Limited: Device failure causes only limited or insignificant impact on the behavior and performance of the device/implementation.

You express functionality and impact in `@card.attribute` tags of the following form:

```
* @card.attribute functionality=value
* @card.attribute impact=value
```

In both cases, *value* must be 1, 2, or 3. CODE EXAMPLE 4-4 shows an example. TABLE 4-3 shows that this case's computed failure severity is 4 - Very Low.

CODE EXAMPLE 4-4 Example Severity Attributes

```
* @testcase
* ...
* @card.attribute functionality=2
* @card.attribute impact=3
```

CHAPTER 5

Writing Online Documentation

This chapter describes the online documentation that you must include with the test packs you develop.

A test pack has four levels of increasingly detailed online documents. TABLE 5-1 summarizes the document types. The graphical harness displays a document when a tester selects a test pack, package, case, or class in the test tree and clicks the Documentation tab.

TABLE 5-1 Types of Test Documentation

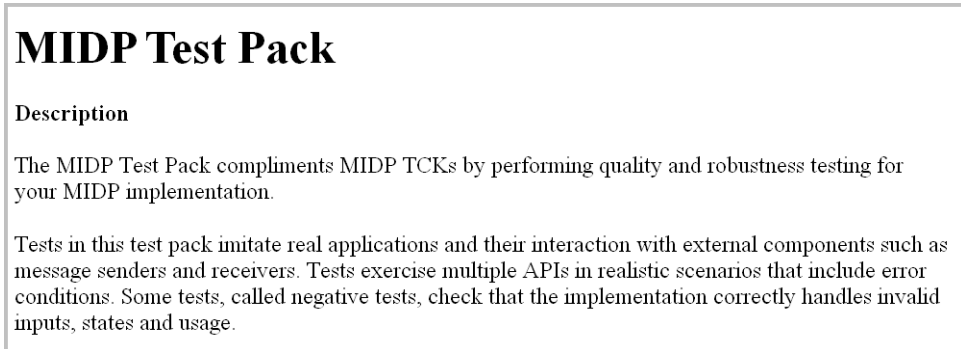
Test Documentation	Description
Test pack	The `testsuite.html` file gives an overview of the test pack. It is described in "Documenting a Test Pack" on page 45.
Test package	The `package.html` file describes the test classes or subpackages in the same directory. It is described in "Documenting a Test Package" on page 46.
Test class	Javadoc tool comment blocks in the source code describe each test class. By default, building a test pack generates HTML file from these comments. These comments are described in Chapter 4.
Test case	Javadoc tool comment blocks in the source code describe each test case. By default, building a test pack generates HTML file from these comments. These comments are described in Chapter 4.

Documenting a Test Pack

Users access test pack documentation from the harness Documentation tab.

Provide overview documentation of a test pack by creating a `testsuite.html` file in the test pack's top directory (the same directory that contains `testsuite.info`). The Java Device Test Suite does not define the content of this file. Its form and content are up to you. FIGURE 5-1 shows one example of test pack documentation.

FIGURE 5-1 Sample Test Pack Documentation



Documenting a Test Package

You must supply a `package.html` file in client functional directories. A non-functional directory is one whose only purpose is to create a unique package name. For example, `com/` and `sun/` in the built-in and sample test packs are non-functional. The harness displays a `package.html` file when a tester selects a package in the test tree and clicks the Documentation tab. By default, the build system recursively scans your test pack's directories and finds all `package.html` files. If you require finer control, see Chapter 26.

Server directories are not required to have `package.html` files, but you can provide them to comply with Java coding guidelines (see `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`). The harness does not display server `package.html` files.

Use only simple HTML commands in `package.html` files. Do not use style sheets, URL links, or browser-specific commands.

Include any information in a `package.html` file that is helpful to testers or administrators who select the package in the graphical user interface and click the Documentation tab. For example, give an overview of the tests in the package to help a tester or administrator decide if exploration of the package's contents is likely

to be worthwhile. Try to answer this question: If you were a tester exploring this package, what would you want to know? If you have no information for testers or administrators, put “No documentation for this package” in the file.

CHAPTER 6

Writing the `testsuite.info` File

The `testsuite.info` file at the top of the test pack directory is a properties file. It contains a list of properties and values. The harness uses some of these properties. You can define additional properties that your tests require to run as desired. "Obtaining a Property Value" on page 61 describes how to retrieve a property value in a test. For each property, you can define its value, its visibility to the harness users, and, if it is visible, assign attributes to the property value, and determine where the property appears in the harness.

Note – Some examples in this chapter have comment lines rendered in *italic* type. These describe the subsequent lines to the readers of this guide. Real files have comments that are helpful to future developers.

File Format and Syntax

You can create `testsuite.info` with any text editor. A `testsuite.info` file can contain any UTF-8 character.

The `testsuite.info` file consists of two kinds of lines:

- *name=value pairs*, for example:

```
TestPackName=Sample Runtime Tests
```

- Comments beginning with a pound sign character (#), for example:

```
# Required entries
```

For advanced developers, a more detailed description of the `testsuite.info` file syntax can be found at `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html`.

Note – An underscore replaces space characters in the directory name where the test pack is installed on the Developer Installation, (*installDir*`/admin/shared/testpacks/`). For example, a test pack with the property `TestSuiteName=AAA BBB CCC` appears as `.../admin/shared/testpacks/AAA_BBB_CCC/`.

Specify lines in any order. However, you must observe `testsuite.info` syntax and semantic requirements exactly. No error messages inform you of mistakes. If you make an error, a test, the tester edition harness, or the administrator edition harness exhibits unexpected behavior.

To see example `testsuite.info` files, refer to the sample test packs in *devKitHome*`/tests/`.

Default Values

An entry like `MyPort=8086` specifies that 8086 is the default value for `MyPort`. Specify a default value that is highly likely to work at your site or sites where a test pack installation will take place. An administrator or user can override the value you specify (subject to the property's scope as described in "Scope" on page 50) but your goal is to provide defaults that do not normally require an override.

Scope

A property's *scope* defines its visibility to administrators and testers through the harness. A property can have the following scopes:

- **advanced –** Follows the default scope properties in the Configuration Editor's center pane. Use this scope for properties whose default values are directed at users with an extensive understanding of the effect of a change.
- **hidden –** The property is hidden and invisible to both administrators and testers. It can, however, be read by a test. For more information, see "Obtaining a Property Value" on page 61.

A property has an implicit default scope. The property default scope is visible and its value is editable in the Java Device Test Suite Configuration Editor. You do not need to specify the scope to apply the default. Use the default scope for properties that a tester will more commonly access.

Note – Do not define a hidden property without a default value.

Note – Do not give a property more than one scope or define a property more than once.

The entries in CODE EXAMPLE 6-1 show how to specify property scopes.

CODE EXAMPLE 6-1 Specifying Property Scopes

```
# default scope implied
MyInOutPort=8088

# advanced scope
MyOutPort=8087
MyOutPort.scope=advanced

# hidden scope
MyInPort=8086
MyInPort.scope=hidden
```

Read-only Properties

To define a property that is visible to users but not editable, code it as shown in CODE EXAMPLE 6-2.

CODE EXAMPLE 6-2 A Read-only Property

```
# DeployMode value is visible, but not editable
DeployMode=HTTP
DeployMode.readonly=true
```

Property Value Validation Attributes

The harness can validate a property value set by a tester against attributes you define for the property. The attributes are listed in order of priority in case there is a conflict, such as a `type` of `boolean` and `max` of 6. Conflicting lower-priority attributes (in this example, `max`) are ignored.

- `.type` – Ensures that the property value entered in the harness is of the correct type. The attribute can be set to `string`, `int`, `boolean`, `float`, and `file`. The `string` type is the default `type` and you do not need to specify it explicitly.
- `.min`, `.max` – Ensures that the property value entered in the harness is within a specific range. This attribute only applies to a property `type` attribute `int` or `float`. If you do not add this attribute to a property, the corresponding non-inclusive value is implicitly applied. For example, by default

`Integer.MAX_VALUE` and `Integer.MIN_VALUE` are applied to an attribute `type=integer` value, and `Float.MAX_VALUE` and `Float.MIN_VALUE` are applied to an attribute with the value `type=float`.

- `.values` – Ensures that the property value entered in the harness is selected from a list you create with this attribute setting. The list appears as a drop-down box in the harness. This attribute only applies to a property value of type `string`. If you do not add this attribute to a property, the harness user can enter any value that does not violate a previously listed attribute setting by default. The value appears in a free text field within the harness.

The entries in CODE EXAMPLE 6-3 show how to specify property attributes.

CODE EXAMPLE 6-3 Specifying Property Attributes

```
# default scope implicit for all properties

# TimeOut default value of 60000 can be reset within the predetermined rang
TimeOut=60000
TimeOut.type=int
TimeOut.min=3000
TimeOut.max=120000

# AccLocation default value of North America can be changed by selecting a new location
#from a drop-down list. More info about the locations can be found at regionalLocal.html
AccLocation=North America
AccLocation.type=string
AccLocation.values=North America,South America,Europe,Asia,Africa
```

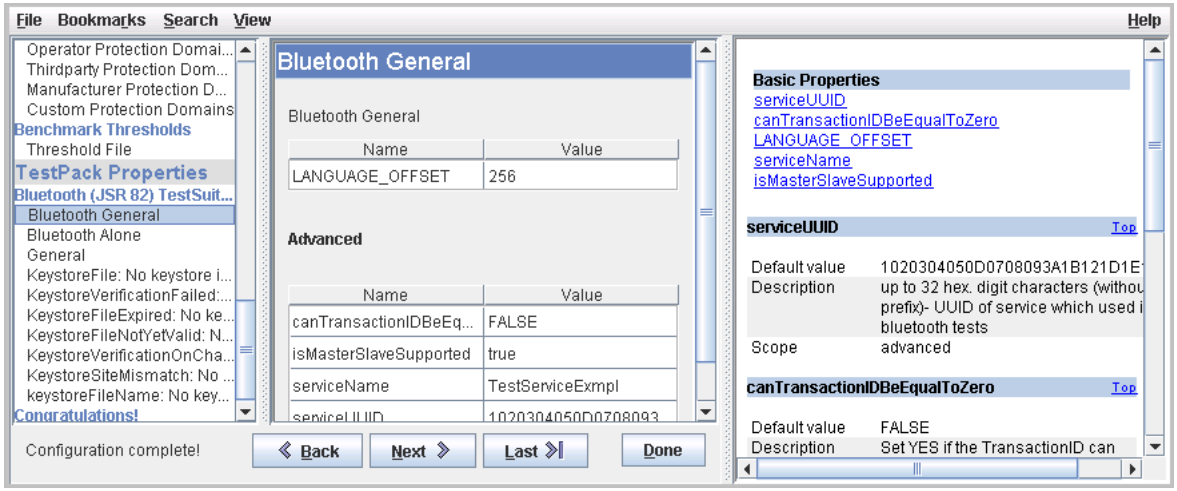
Categories

The default or advanced scope property's *category* determines where the harness accesses and displays a property in the harness's Configuration Editor. The Configuration Editor's left pane lists test packs (bold font entries). Each test pack contains a list of categories. Testers choose a category and the properties assigned to the category display in the center pane. Default scope properties appear at the top of the pane. Advanced scope properties follow.

All test packs have a default category named General. Properties not explicitly assigned to a category are assigned to General.

FIGURE 6-1 shows the Configuration Editor with a selected category in the left pane and its assigned default and advanced properties in the center pane.

FIGURE 6-1 Configuration Editor



For more information about the Configuration Editor, see the Java Device Test Suite online help.

CODE EXAMPLE 6-4 shows how to specify categories for default scope properties. You can specify a text description for a category. If the category name includes a space character, you must escape it with a back slash "\" as the example shows. If you do not provide a text description, the category name is used. For example, FIGURE 6-1 has "Bluetooth General" for both the category name (blue band) and the category description below it.

CODE EXAMPLE 6-4 Specifying Default Scope Property Categories

```
# implicitly goes in General category
MyInOutPort=8088

# explicitly goes in Network Tests category
MyOutPort=8087
MyOutPort.category=Network Tests

# optional description text for Network Tests category
Network\ Tests.CategoryDescr=Properties for Network Tests
```

CODE EXAMPLE 6-5 shows how to specify categories for advanced scope properties.

CODE EXAMPLE 6-5 Specifying Advanced Scope Property Categories

```
#implicitly goes in General category
MyOutPort=8087
MyOutPort.scope=advanced
```

CODE EXAMPLE 6-5 Specifying Advanced Scope Property Categories *(Continued)*

```
# explicitly goes in Network Tests category
MyOutPort=8087
MyOutPort.scope=advanced
MyOutPort.category=Network Tests
```

There is no point to assigning a category to a property whose scope is hidden.

Online Documentation for a Property

You can supply a short description for a property that appears in the harness Configuration Editor's More Info (right) pane next to the Description title (as shown in FIGURE 6-1). To add the short description, follow the property definition with a `.doc` line attribute, for example:

```
MyPort=8086

MyPort.doc=Specify an IP port that the test server can use.
```

Supply a `.doc` attribute entry for every property whose scope is *not* hidden.

Path Names

Many property values are path names. The path name separator is the slash (`/`) character.

`${TS_DIR}` Reserved Word

The reserved word `${TS_DIR}` can be used in path names (and must not be used for anything else).

In a property you define, `${TS_DIR}` refers to the root of the test pack's directories *in the Relay file system*. If your test pack includes tests that have client and server parts, you can use `${TS_DIR}` to give server parts access to test pack files that the test pack installer copies to the Relay file system. For example, suppose a server part needs a sample media file that you have created in *packWorkDir*`/test_data/samples/`.

1. **Set the predefined** `TestServerResources` **property to** `test_data/`**.**

   The test pack installer copies the directories named in `TestServerResources` to the Relay file system.

2. **Define another property, for example,** `SamplesForServer`**, and set it to** `${TS_DIR}/test_data/samples/`**.**

3. **Write the client part to retrieve** `SamplesForServer` **and pass its value to the server when it (the client) calls** `Runner.launchServer()`**.**

   When the client (running on the test device) asks for the value of `SamplesForServer`, (see "Obtaining a Property Value" on page 61) it receives the absolute path to `test_data/` in the Relay file system.

4. **Write the server part to use the value passed to it to find the** `samples/` **directory.**

   "Writing Network Tests" on page 65 describes the details of client and server interaction.

Note – The notation `${TS_DIR}` supersedes `$(TS_DIR)`. Although the old form is supported for backwards compatibility and is used in some of the sample `testsuite.info` files, you should use the new notation.

Required Properties

A `testsuite.info` file must include predefined properties that the harness uses to load and display the test pack components. The following properties are required:

- `TestSuiteID` – The test pack's ID. The harness uses this property to identify test packs. A new test pack's TestSuiteID must be different from the TestSuiteIDs of all installed test packs. If a to-be-installed test pack's TestSuiteID matches the TestSuiteID of an installed test pack, the test pack installer assumes that the installed test pack is to be updated (see "Test Pack Identifiers" on page 89). It is recommended to construct the TestSuiteID value in the style of dot-separated namespace identifiers, such as `com.mycompany.sip`. Be sure the value does *not* have leading or trailing space characters.
- `TestSuiteName` – A Java Device Test Suite 1.4 legacy property. The test pack installer uses the `TestSuiteName` property as the name of the directory containing the test pack. A new test pack's `TestSuiteName` must be different from the `TestSuiteNames` of all installed test packs.
- `TestPackName` – The name of the test pack. This is the name that the harness displays after test pack installation. A new test pack's `TestPackName` must be different from the `TestPackNames` of all installed test packs. When installing a test pack whose pack name contains spaces, the `TestPackName` replaces spaces with underscores. For example, `Sample Runtime Tests` changes to `Sample_Runtime_Tests`.

- `TestSuiteType` – The test pack type, `Runtime`, `Performance`, or `OTA` (benchmark tests were formerly called performance tests and the name has been retained for backwards compatibility).
- `TestClassesDir` – The test pack binaries directory.
- `TestSourcesDir` – The test pack source directory.
- `TestDocDir` – The test pack documentation directory.
- `TestServerResources` – The comma-separated list of test pack source directories on the server side (required only where a server side exists).

The preceding properties must have a scope of hidden (see "Scope" on page 50).

Benchmark and OTA test packs require additional properties in `testsuite.info`. For information about these additional properties, see the sample files.

Optional Properties

A `testsuite.info` file might include other predefined properties that the harness uses to load and display the test pack components. The following properties are optional:

- `TestServerSourcesDir` – This property contains a colon-separated list of one or more server-side source roots. The value was formerly hard coded as `src/server`, which remains the default.

  The root represents a directory several levels above the server-side Java source files. The directory containing source files is determined by the path implicit in the namespace of the package. For example, in the Developer's Kit runtime examples, the `tests/runtime/testsuite.info` file defines `TestServerSourcesDir` to include the root src/server. The actual server-side Java source (for example, the file `GenericRequestThread.java`) is found in the directory

  ```
  tests/runtime/src/server2/com/sun/samples/network/server/http/
  ```

  The file `GenericRequestThread.java` includes the line

  ```
  package com.sun.samples.network.server.http;
  ```

  which shows how the namespace maps to the location of this source file in the file system.
- `MIDletDir` – This property is a pointer to the directory containing the JAD files specifying the test MIDlets for an OTA test pack or the partner MIDlets in a runtime test pack that has distributed tests. For an example of using this property, see `tests/ota/testsuite.info`.

- `MIDletSourcesDir` – This property is a pointer to the directory containing the sources for the test MIDlets for an OTA test pack. For an example of using this property, see `tests/ota/testsuite.info`.
- `TestSuiteVersion` – The test pack version identifier assists in test pack management and is displayed by the harness's Help -> About the Java Device Test Suite. Form the version number from digits separated by periods, following your organization's conventions. For example, `1.2` or `1.2.2`.

  You can specify the test pack version in this property or in the file `testpack.version.properties`, which is described in "Test Pack Versioning Alternative" on page 109.

Note – Do not use both versioning mechanisms. `TestSuiteVersion` in `testsuite.info` overrides `testpack.version.properties`.

  While you are developing or modifying a test pack, you must prefix the version number with the characters `++`, for example:

```
TestSuiteVersion=++1.4
```

  These characters ensure that when you launch the harness, it synchronizes templates with the latest changes you have made in the test pack, for example, new properties. When you have finished development, remove the `++` characters, increment the `TestSuiteVersion` value by an amount of your choosing, and run `ant pack` to create the installable test pack.

- `TSPermissions` – If your test pack requires access to protected APIs (as defined by the MIDP 2.0 security model), you must include the `TSPermissions` property in `testsuite.info`. For the value of the property, name all permissions required by all tests in a comma-separated list. For example (ignore line breaks):

```
TSPermissions=javax.microedition.io.Connector.http
,javax.microedition.io.Connector.sms,javax.microedition.apdu.sat,j
avax.microedition.payment.process
```

  If you specify `TSPermissions`, you must include the permission `javax.microedition.io.Connector.http` in the list. The MicroAgent in each bundle uses the HTTP API to communicate with the Relay. If you do not specify `TSPermissions`, the harness automatically requests `javax.microedition.io.Connector.http`.

  If you specify `TSPermissions`, hide it from administrators and testers with this line:

```
TSPermissions.scope=hidden
```

  The harness uses the `TSPermissions` property to reduce the number of permission requests it includes in test bundles. The requested permissions in a bundle are the intersection of the permissions selected in the configuration's

Security Permissions questions and the value of the `TSPermissions` property in the test pack whose tests are in the bundle. In other words, the harness uses `TSPermissions` properties to filter out irrelevant user-selected permissions.

CHAPTER 7

The `build.properties` Files

When you build a test pack, the build is controlled by defaults specified in *devKitHome*`/tests/common/build/build.properties` and overridden by values you specify in *devKitHome*`/tests/`*yourTestPack*`/build/build.properties`. Initially, there are no overrides in your test pack's `build.properties` file, so all defaults specified in the common `build.properties` file are in effect.

The common `build.properties` file has comments that describe each property and its values. The default values are likely to work well for you, but you should override at least one of them. By default, generated files contain a Sun Microsystems copyright. You should change this to copyright text appropriate for your organization. For example:

```
doc.copyright=\
\ Copyright 2007 My Company. All rights reserved. \n\
\ Use is subject to license terms.
```

CHAPTER 8

Using Common Services

All tests can use services provided by the singleton class `com.sun.TestBeans.Agent.TestLoader.Runner`. For a complete description of this class, see *devKitHome*`/docs/testapi/index.html`.

The services are described in the following sections:

- Obtaining a Property Value
- Learning if a Case is Selected
- Logging

Obtaining a Property Value

To obtain the value of a property defined in the `testsuite.info` file or a source file, call `Runner.getProperty()`. For details, see the Test API documentation. For examples of usage, see the sample tests.

If a given property is defined in multiple places, the precedence rules described in Chapter 16 determine which value `Runner.getProperty()` returns.

Learning if a Case is Selected

To learn if a particular test case is to be run, call `Runner.isSelected()`, passing in the name of the case. For details, see the Test API documentation. For examples of usage, see the sample tests.

Logging

The `Logger` class provides multi-level log messaging, creating tagged log messages that represent a hierarchy of importance. The least important level is `TRACE`, and the most important level is `FATAL`. The importance tags in log messages enable the MicroAgent to select the messages it returns to the Relay based on the importance level set by the tester in the harness.

The `Level` and `Logger` classes define the logging facility. The `Level` class documentation gives guidance for choosing the level appropriate for a message.

The following sample gives one syntax for setting log message levels:

devKitHome`/tests/runtime/src/client/com/sun/samples/automated/Logging.java`

Given a `Logger` named `log`, you can alternatively use the methods `log.trace()`, `log.debug()`, and so on.

Other sample tests give examples of typical logging use.

CHAPTER 9

Writing Runtime Tests

This chapter describes the basics of writing runtime tests. It contains the following sections:

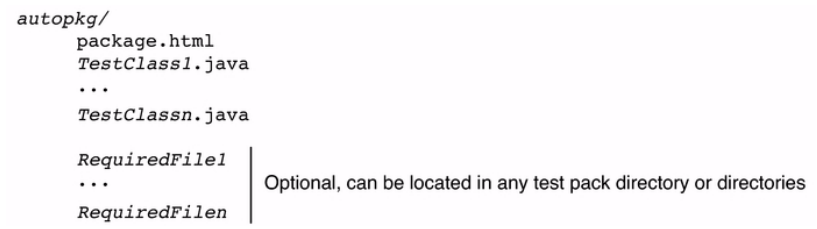
Test pack development is an iterative, non-linear process that varies according to your local practices and preferences. As you read the following sections, remember that when developing tests you might perform some steps many times, and you might perform the steps in a different sequence than they are described in this chapter. You might also choose to build and test your developing test pack more frequently than this chapter suggests. If you use a source code control system, you might have to modify some instructions, such as "make the file writable."

Writing an Automated Test

Although not required, it is a good practice to create test packages that contain only automated test classes or only interactive test classes. This practice creates a smaller and easier-to-navigate test tree when users create sessions that contain only automated or only interactive tests, because entire packages are removed from the tree. The sample tests in this chapter follow this practice (which also makes it easier for you to find an automated or an interactive test example). However, you can create test packages that contain both automated and interactive test classes.

The directory that corresponds to an automated test package contains the files shown in FIGURE 9-1.

FIGURE 9-1 Automated Runtime Test Directory Contents

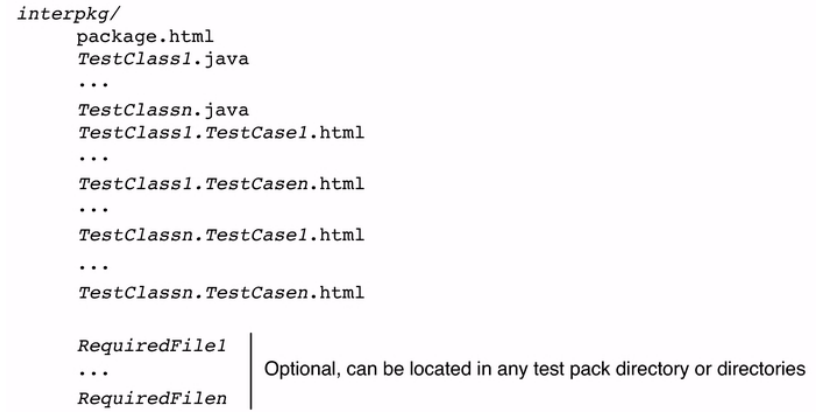


An automated test implements `AutomatedTest`, an interface that is documented in *devKitHome*`/docs/testapi/index.html`. For a sample automated test, see *devKitHome*`/tests/runtime/src/client/com/sun/samples/automated/SampleAutomatedTest.java`.

Writing an Interactive Test

The directory that corresponds to an interactive test package contains the files shown in FIGURE 9-2. A test package can have any number of classes and a test class can have any number of cases.

FIGURE 9-2 Interactive Runtime Test Directory Contents



An interactive test extends `InteractiveTest`, an abstract class documented in *devKitHome*`/docs/testapi/testapi/index.html`. For a sample interactive test, see *devKitHome*`/tests/runtime/src/client/com/sun/samples/interactive/SampleInteractiveTest.java`.

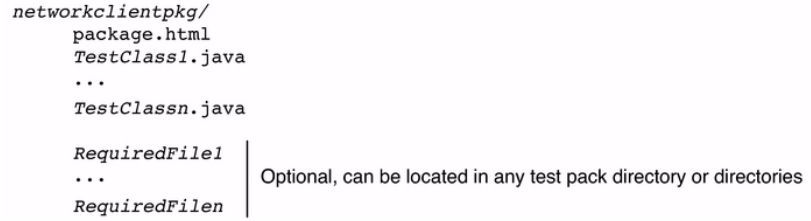
Writing Network Tests

As described in "Network Tests" on page 7, network tests have a client part that runs on the test device and a server part that runs on the Relay. These parts are implemented in different directories.

Writing the Client Part

FIGURE 9-3 shows the directory and files that corresponds to a network client test package.

FIGURE 9-3 Network Client Test Directory Contents



The following sections describe how to create one network client test package containing one test class. Repeat the instructions to create additional packages. Use subsets of the instructions to add classes, or required files, or both to a package. The build process automatically updates the `packages.list` file.

A network client test implements `AutomatedTest`, an interface documented in *devKitHome*`/docs/testapi/testapi/index.html`. For sample network client tests, see the following:

- *devKitHome*`/tests/runtime/src/client/com/sun/samples/network/client/http/SampleHttpReadClient.java`, a client that receives a single HTTP message
- *devKitHome*`/tests/runtime/src/client/com/sun/samples/network/client/http/SampleHttpWriteClient.java`, a client that sends a single HTTP message

To launch its server part, a client calls `Runner.launchServer()`, passing the server implementation class name and arguments that will be passed to the server's `init()` method. `launchServer()` sends an HTTP message to the Relay, which loads the test server. You define the arguments, but one of them is typically the port on which the server listens for messages from the client. The common way to establish this port is to define it as a client property.

The `launchServer()` method returns an integer. If the return value is greater than 0, it is the server port number. If the returned value is 0 or a negative number it indicates an error code returned by the server. For more information about the error codes, see the following API documentation:

- `com.sun.midp.testmanager.AbstractTestServer.init(String[] params)`
- `com.sun.midp.testmanager.AbstractTestServer.TEST_SERVER_INIT_FAILED`
- `com.sun.midp.testmanager.AbstractTestServer.TEST_SERVER_LOAD_FAILED`
- `com.sun.TestBeans.Agent.TestLoader.Runner.launchServer(String serverName, String[] serverParams)`

Writing the Server Part

A server implementation extends `AbstractTestServer`, which is documented in *devKitHome*`/docs/test-server-apis/index.html`. For examples of network test servers, see the following:

- *devKitHome*`/tests/runtime/src/server/com/sun/samples/network/server/http/SampleReadTestServer.java`, a server that sends a single HTTP message.
- *devKitHome*`/tests/runtime/src/server/com/sun/samples/network/server/http/SampleWriteTestServer.java`, a server that reads and verifies a single HTTP message. This server uses `BaseHTTPServer.java` in the same directory.

To obtain the value of a property defined in the `testsuite.info` file or in a test class source file, use `AbstractTestServer.getResourceHelper().getProperty()`. If a property is defined in multiple places, the precedence rules described in Chapter 16 determine which value `getProperty()` returns.

Writing Push Tests

A push test uses or tests a device's implementation of the push technology defined in the MIDP 2.0 (JSR 118) `javax/microedition/io/PushRegistry` API. Push technology defines a way for a MIDlet to be awakened by an alarm from the device's clock or an incoming network connection. See the JSR 118 specification for details and the World Wide Web for introductory articles.

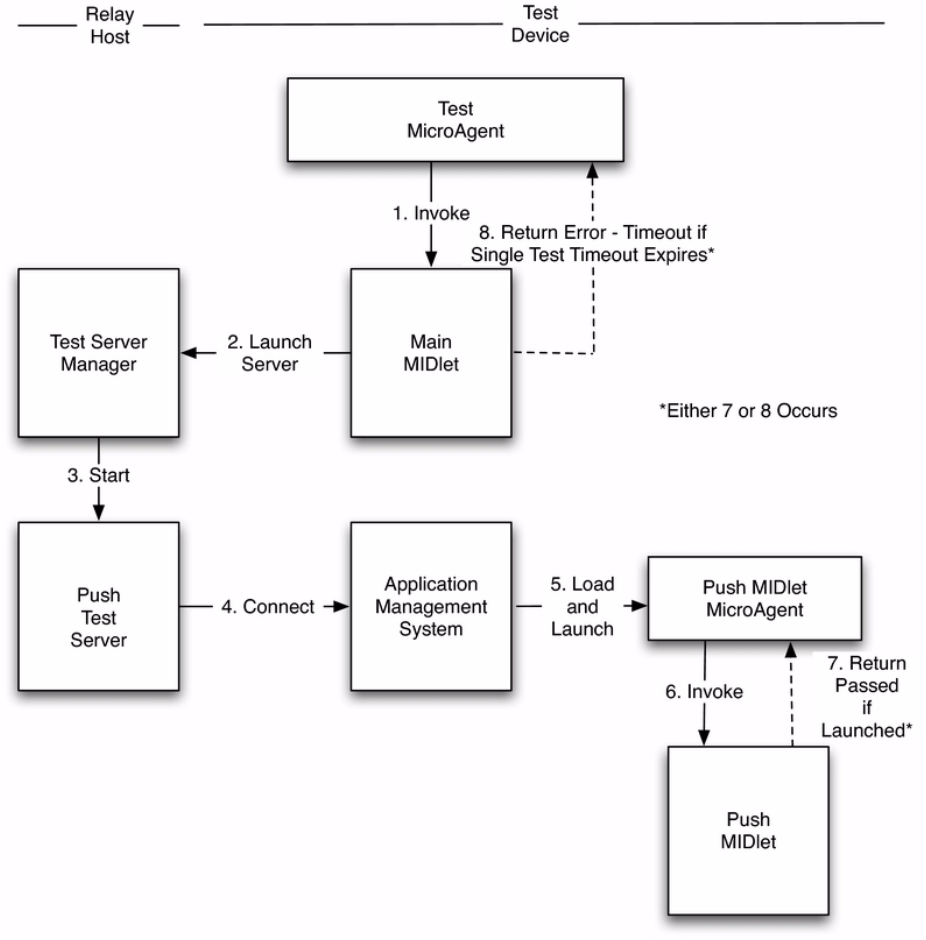
This section describes how to write connection-based push tests (hereinafter simply called “push tests”). Push technology can also be useful in other kinds of tests. For an example, see `registerAlarm.java` and `AlarmMidlet.java` in *devKitHome*`/tests/runtime/src/client/com/sun/samples/network/client/pus/push/`.

You must be familiar with “Writing Network Tests” on page 65 to use this section successfully.

Architecture of a Push Test

Similar to a network test, a push test has client and server parts. However, the client is composed of two MIDlets, called the main MIDlet and the push MIDlet. For downloading, the main and push MIDlets are bundled together but they are separate applications. These components and their interactions are shown in FIGURE 9-4.

FIGURE 9-4 Connection-based Push Test Components and Interactions



The components interact as follows:

1. The test MicroAgent (see "Automated Tests" on page 3) invokes the main MIDlet.
2. The main MIDlet is primarily an initializer. It directs the test server manager to start the push test server, registers the push MIDlet if dynamic registration is being tested, and returns `WAIT_FOR_PUSH_RESULT`. This return value causes the main MIDlet to periodically check the Record Management Service store for a value that indicates the push MIDlet has returned a result. If the Single Test Timeout specified in the test run configuration expires, the main MIDlet returns an Error - Timeout status. If the main MIDlet finds the value created by the push MIDlet, it exits.
3. The test server manager starts the push test server.

4. The test server opens a connection to the device.
5. The device's application management system (which implements the push registry) loads and launches the push MIDlet (and its MicroAgent), which has been registered and associated with the server's connection.
6. The push MIDlet's MicroAgent invokes the push MIDlet.
7. If all steps have been successful, the push MIDlet notifies its MicroAgent that the test has passed.
8. If, due to a failure, the push MIDlet is not launched, the main MIDlet wakes from its sleep after the configuration's Single Test Timeout value expires and notifies its MicroAgent that the test has failed.

Writing the Client Package

A push client package is identical to a network client package, except that it also has at least one *pushMIDlet*`.java` file. See FIGURE 9-3 for details.

Writing a Main MIDlet

A push test main MIDlet implements AutomatedTest, an interface whose documentation is accessible from *devKitRoot*`/docs/test-api/index.html`. See "Automated Runtime Test Directory Contents" on page 64 for a description of an automated test class.

There are two example main MIDlets in *devKitRoot*`/tests/runtime/src/client/com/sun/samples/network/client/push/`:

- `BasicStatic.java` - registers the push MIDlet statically by an entry in the test bundle's JAD file.
- `BasicDynamic.java` - registers the push MIDlet dynamically by calling `pushRegistry.registerConnection()`.

The examples are otherwise logically identical.

A main MIDlet's comment block (see Chapter 4**)** must contain the equivalent of the following line to ensure that the push MIDlet (and all classes it requires) is included in the same test bundle as the main MIDlet (ignore the line break):

```
* @card.requires
com/sun/samples/network/client/push/PushMIDlet.class *
```

A push test's main MIDlet comment block must include a line that declares the push MIDlet in the test bundle JAD file in accordance with the MIDP 2.0 specification. For example:

```
* @card.specialproperty <jad>.1=MIDlet-2: \
*PushMIDlet,,com.sun.samples.network.client.push.PushMIDlet
```

For more information on `@card.requires` and `@card.specialproperty`, see "`@card.specialproperty`" on page 41.

If the test registers the push MIDlet statically, the comment block must also include a line that causes the registration information to be included in the JAD file in accordance with the MIDP 2.0 specification. For example:

```
* @card.specialproperty <jad>.2=MIDlet-Push-1: socket://:5775, \
*com.sun.samples.network.client.push.PushMIDlet,\
*$IPFILTER("0123456???*")
```

If a main MIDlet has more than one push MIDlet, provide `@card.requires` and `@card.specialproperty` comments for each push MIDlet.

Writing a Push MIDlet

A push MIDlet can be automated or interactive. For an example of an automated push MIDlet, see *devKitRoot*`/tests/runtime/src/client/com/sun/samples/network/client/push/PushMIDlet.java`.

For an interactive example, see *devKitRoot*`/tests/runtime/src/client/com/sun/samples/network/client/push/AlarmMidlet.java`.

Writing a Server

A push server is identical to a network server described in "Writing the Server Part" on page 66. For an example, see *devKitRoot*`/tests/runtime/src/server/com/sun/samples/network/server/push/SamplePushTestServer.java`.

This server has a simple "one-shot" logic. It does not have a "conversation" with the main MIDlet. When the test server manager calls the server's `init()` method, the method creates a socket connection to the device, writes a 0 to the socket, and closes the socket. If your server needs are as simple, you can re-use reuse this server. The network examples have more elaborate server samples.

Using $IPFILTER

The $IPFILTER is a macro that can be used in JAD file values specified with a special property attribute. It is useful when the user needs to register a push MIDlet statically. For example,

```
MIDlet-Push-1: socket://:5775, com.sun.samples.network.client.
  push.PushMIDlet, $IPFILTER("0123456???*")
```

In this example, the IP mask is specified using $IPFILTER. with expands depending on the IP address of the relay (as it seen from the device) according to the rule specified by the filter.

The syntax is

```
$IPFILTER("string")
```

where string is the filter. It is composed of the characters 0,1,2 ...,9,A, .. ,D,E,*,?

The *N*th character of the result is the *K*th character of he IP address, where *K* is found in the *N*th position of the filter or "*" or "?" respectively if "*" or "?* is found in the *N*th position of the filter. For example, if the second character of the filter is "E", the second position of the result is the *E*th (last) character of the IP address. If the second character of the filter is "?", the second character of the result is "?".

Note – The IP address is considered to be in 15-digit form (*nnn.nnn.nnn.nnn*, not *nn.n.nnn.nn*). The digits of the IP address are numbered from 0 to E.

Example: Suppose *IP=AAA.BBB.CCC.DDD* is the IP address of the relay as seen from the device. Each character of the IP address is used to calculate an arithmetic mean with the corresponding character of the filter.

The special characters * (asterisk) and ? (question mark) stand for themselves.

Example: If the IP address of the relay is 129.159.125.101 and the filter is $IPFILTER("0123456???*"), the resultant string is 129.159???*.

Example: Using the same IP address for the relay, the filter $IPFILTER("EDCBA9876543210") expands to the string 101.521.951.921.

CHAPTER 10

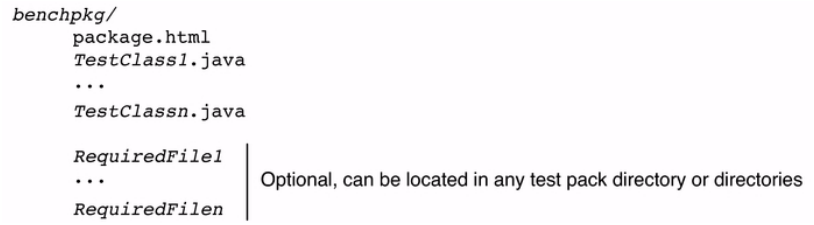
Writing Benchmark Test Packages

This chapter describes how to create a benchmark tests. It contains these sections:

Benchmark Test Directory Structure

FIGURE 10-1 shows files in a directory that correspond to a benchmark test package. It is identical to a runtime test directory.

FIGURE 10-1 Benchmark Test Directory Contents



Benchmark Test Types

Java Device Test Suite has two types of benchmark tests:

- **System Load Test –** Determines an approximation of the percentage of processor time a constant-time operation takes, for example, playing an audio file. The core of a system load test case runs in parallel with a calibration thread that the harness starts. Having previously run the calibration thread without competition from the test, the harness computes how much less work the calibration thread accomplishes when run in parallel with the test. If, for example, when run in parallel with the test, the calibration thread achieves 40% of the work it achieved when run alone, the test system load is 60%.
- **Unit Rate Test –** Sampling over a fixed interval, determines how many operations, such as redrawing a screen, a test can complete. The result is an array of frame rates. The test notifies the harness every time it completes a unit of work. The harness returns a parameter that tells the test to perform another iteration or stop.

Writing a System Load Test

A system load test extends `ComprehensiveBenchmarkTest`, a class documented in *devKitHome*`/docs/testapi/testapi/index.html`. For a sample system load test, see *devKitHome*`/tests/benchmark/src/client/com/sun/samples/benchmark/testsystemload/MediaPlayerTest.java`.

For realistic system load example, see *installDir*`/admin/shared/testpacks/Benchmark_TestSuite/src/client/com/sun/benchmark/scenarios/mediaPlayer/MediaPlayerTest.java`.

Writing a Unit Rate Test

A unit rate test extends `ComprehensiveBenchmarkTest`, a class that is documented in *devKitHome*`/docs/testapi/testapi/index.html`. For a sample unit rate test, see *devKitHome*`/tests/benchmark/src/client/com/sun/samples/benchmark/testUnitRate/GraphicsTest.java`.

For realistic unit rate examples, see the directories in *installDir*`/admin/shared/testpacks/Benchmark_TestSuite/src/client/com/sun/benchmark/scenarios/`.

However, ignore the `mediaPlayer` directory because it contains a system load test.

CHAPTER 11

Developing an Over-the-Air Test Package

This chapter describes how to create an over-the-air (OTA) provisioning test package. It contains the following sections:

- OTA Test Pack Development
- Writing an OTA Test
- Writing Application Files
- Additional Facilities for Interactive OTA Tests

Test pack development is an iterative, non-linear process that varies according to local practices and preferences. As you read the following sections, remember that when developing tests you might perform some steps many times, and you might perform the steps in a different sequence than they are described in this chapter. You might also choose to build and test your developing test pack more frequently than this chapter suggests. If you use a source code control system, you might have to modify some instructions, such as “make the file writable.”

OTA Test Pack Development

OTA test pack development is quite different from runtime or benchmark test pack development. OTA test source files are essentially empty except for Javadoc tool class and case documentation and other markup (see Chapter 4). The Java Device Test Suite provides all OTA test code, which is driven by properties you specify. The test code does *not* run on a test device. It runs on the Relay.

In OTA tests, the Relay acts as a provisioning server. It performs these services:

- Upon test device request, sends HTML or WML with a link to a JAD file specifying a MIDlet.

- Upon test device request, sends the JAD file.
- Upon test device request, sends JAR file.
- Receives the installation status code sent by the test device after installation; for semi-automated OTA tests: compares this code vs. expected one and forms the result.
- Receives the result sent by test MIDlet (this is another specific mechanism: the installed MIDlet sends a result string to specific address on the relay, relay compares this string against the expected, and returns the test result.)
- Communicates with the test harness. It sends results, knows the current test ID and its properties, sends commands to display the test description, and so forth.

Each test has at least one associated MIDlet that the user downloads to the test device. You write these test MIDlets.

Program logic is different in OTA tests compared to other test types (where the test logic runs predominately within the test device). In OTA tests, the program logic is an interplay between the following.

1. The test MIDlet, installed successfully (or not) onto the test device, and executed in the device.
2. The installation procedure. The tester reads test instructions and interacts with the test device and the MIDlet.
3. The AMS, a part of the test device that manages the installation and launching of the MIDlet. For example, the AMS sends an installation status report to the relay after MIDlet installation.

To develop an OTA test pack, you must also know how to modify Ant build scripts.

Note – Compared to earlier releases of JDTS, OTA test development is different after JDTS version 2.0. However tests, written for earlier releases can run on a 2.0 harness with minor modifications. Follow the instructions in this chapter to develop new OTA tests after JDTS version 2.0.

Writing an OTA Test

An OTA test has two sets of files:

- The *test package files* document the test, provide user instructions, and direct the execution of the generic OTA servlet. These files must be created in a subdirectory named by the `testsuite.info` file's `TestSourcesDir` property. The sample test files are in *devKitHome*`/ota/src/client/`.
- The *application files* implement a MIDP application (MIDlet) that the user downloads to the test device when running the test. These files must be created in the directory named by the `testsuite.info` file's `MIDletSourcesDir` property. The sample application files are in *devKitHome*`/ota/src/apps/`. Multiple tests can share the same test MIDlet.

  See the property `MIDletSourcesDir` in the file *devKitHome*`/ota/testsuite.info` for an example of good paths to MIDlet source files. It is important to structure the paths to avoid namespace conflicts, and this is reflected in the directory structure of the OTA examples.

You can create the file sets in any order. This section describes how to write the files in a test package. "Writing Application Files" on page 80 describes how to write application files.

Writing OTA Source Files

An OTA test source file is only a container for online test documentation and properties. It has no executable statements. The OTA servlet and, for some tests, the associated test MIDlet, do the work of the test. For a sample interactive test, see *devKitHome*`/tests/ota/src/client/com/sun/samples/interactive/InteractiveActive.java`. Interactive and semi-automated test source files have the same format.

Security Certificates for a Test Class

Tests that use protected APIs must be granted permission to call methods in those interfaces. This is done by digital certificates in the test device which are associated with protection domains.

OTA test developers must specify the protection domain for the users of applications that use protected APIs. The available domains are specified in the MIDP specification.

Use the following syntax in a source file comment (see "Test Class and Case Comment Blocks" on page 29) to make a domain available to testers:

```
* @card.property SecurityDomain=DomainID
```

The `SecurityDomain` property value is the domain name (*DomainID*).

The *DomainID* is one of four OTA pre-defined domains or a custom domain (described later in this section):

- `Untrusted`
- `Manufacturer`
- `ThirdPartyTrusted`
- `OperatorTrusted`.

For example:

```
* @card.property SecurityDomain=ThirdPartyTrusted
```

The preceding example means the OTA MIDlet packs associated with this test case are signed with a key setup user interface (using the Manage Keystores command and corresponding Configuration Editor pane) for the specified protection domain, so that the MIDlet suite are associated with this protection domain in the device where the MIDlet is installed.

In addition to the pre-defined domains previously mentioned, you can create custom domains. Custom domains map to specific certificates in their respective keystores. Custom domains as well as the default domains must be made apparent to the tester. The tester enters the domains in the harness Configuration Editor in the left pane under Over The Air Test MIDlets. Custom domain names have the following restrictions:

- Valid character entries are Latin letters, decimal digits, hyphens (-), underscores (_) or blank spaces ( ).
- Custom names must not start or end with blank character space.

For more information about protection domains, certificates, and keystores, see the *Java Device Test Suite Test Notes* and the administrator harness online help.

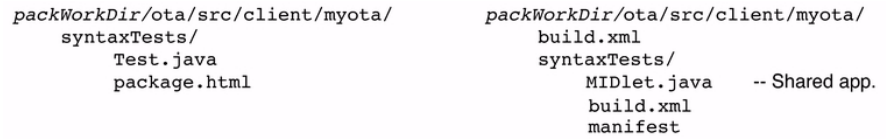
Writing Application Files

Every OTA test case has an associated application (MIDlet, technically a MIDlet suite). The test case's evaluation file typically instructs the user to download the associated MIDlet to the test device.

Directory Structure

FIGURE 11-1 shows, on the left, a directory containing one test class that has two test cases. On the right, the figure shows the application files you must create for this example in which both test cases use the same application (`MIDlet.java`).

FIGURE 11-1 Test and Application File Correspondence



An application source file is a MIDlet, as defined by the MIDP 1.0 and 2.0 specifications. For an example used with an interactive OTA test, see *devKitHome*`/tests/ota/src/apps/samples/interactive/PropExample.java`. Multiple tests can use the same application if their source file comments name the shared application's JAD file and JAR file (which are created by the build script). If a test requires its associated application to perform an operation on the test device, make sure that the application displays information that tells the user to click Passed or Failed in the evaluation window, and that the evaluation window instructs the user to launch the application.

In the same directory as the application, create a `build.xml` file. This file is application specific. See the samples in the subdirectories of *devKitHome*`/tests/ota/src/apps/samples/` for examples. By default, all MIDlet JAD and JAR files are created in the directory specified by `TestMIDletDir`. You can have a MIDlet's JAD file and JAR file created in a subdirectory by specifying a value for *subDir* in the following `build.xml` line:

```
<mkdir dir="${MIDlet.dest.dir}/subDir" />
```

Correspondingly, add *subDir* to all `build.xml` lines that contain `MIDlet.dest.dir`.

The `@card.property` tags in the source file must name the same directory. Here is a hypothetical example line from a `build.xml` file:

```
<mkdir dir="${MIDlet.dest.dir}/mySubDir" />
```

The corresponding source file lines are as follows:

```
* @card.property=JADPath1=mySubDir/TestMIDletJADFile.jad

* @card.property=JARPath1=mySubDir/TestMIDletJARFile.jar
```

In the same directory as the application, create a manifest file named `manifest`. The MIDP 1.0 and MIDP 2.0 specifications define the manifest file format and contents. See *devKitHome*`/tests/ota/src/apps/samples/interactive/color/manifest` for an example.

For MIDlets that use protected APIs, code the permission requests in the MIDlet JAD file and JAR file manifest. To see how these files are built, inspect *devKitHome*`/tests/ota/src/apps/samples/signed/build.xml`.

Application Logging

An application can send timestamped log messages by HTTP. In the test results, these messages are identical to those that runtime tests can create as described in "Logging" on page 62. To generate a log message, add the following line to the application's JAD file:

```
logurl: ${URL_SIMPLE_LOG_RECEIVER}
```

CODE EXAMPLE 11-1 shows how to send a log message to the Relay. The class `JdtsLogLevel` defines the available log level constants, such as `TRACE` and `DEBUG`.

CODE EXAMPLE 11-1 Sending a Log Message

```
...
String logUrl = getAppProperty("logurl");
Connection c = Connector.open(logUrl+"&LogLevel="+myMessageLevel);
HttpConnection hc = (HttpConnection)c;
hc.setRequestMethod(HttpConnection.POST);
OutputStream os = hc.openOutputStream();
os.write(myLogMessage.getBytes("UTF-8"));
os.flush();
os.close();
...
```

Additional Facilities for Interactive OTA Tests

The example in *devKitHome*`/tests/ota/src/client/com/sun/samples/interactive/Interactive.java` has test cases that illustrate several optional features of the interactive OTA test infrastructure:

- **Basic MIDlet installation -** `t01BasicInstallation()`
- **Multiple MIDlets -** `t02InstallMoreThanOneMIDletSuite()`
- **JAR file without a JAD file -** `t03DownloadJarWithoutJad()`
- **Dynamic JAD file attributes -** `t04SetJADPropertyValuesOnRuntime()`
- **Per-test JAD file attributes -** `t05addSpecialJADAttribute()`
- **Advanced facilities of property expansion mechanism -** `t06multilayerPropertyExpansion()`
- **Custom MIME types for JAD files and JAR files -** `t07SetCustomMimeType()`

- **Static custom HTML/WML pages usage -** `t08InstallFromCustomHtmlWmlPage()`
- **Accessing the provisioning server URL from a MIDlet -** `t09getAnotherMIDletSuiteURL()`

PART III Advanced Topics

This part covers the more advanced topics of test pack development. These topics do not necessarily apply to every development project. They cover concepts and procedures that are used in special cases.

Chapter 12 describes how to use the Card File Checker tool to verify card files that have been written manually.

Chapter 13 describes how to update a test pack as part of a test pack development cyclical workflow.

Chapter 14 describes how to point the developer's kit to a different Developer Installation.

Chapter 15 describes how to add security permissions to templates and configurations.

Chapter 16 describes property definition and parameter expansion.

Chapter 17 describes how to edit the file `testpack.archive.properties` to control the installable zip file created by the `pack` build target from a test pack.

Chapter 18 describes how to set up a work directory to handle development of multiple test packs.

Chapter 19 describes how to use a `tpim` script for faster test pack update cycles.

Chapter 20 describes an alternative procedure for specifying the test pack version (the variable `TestSuiteVersion`), using the file `testpack.version.properties`.

Chapter 21 describes the Ant build targets in the common build scripts.

Chapter 22 describes the optional feature definition file.

Chapter 23 describes how to optionally filter tests that are irrelevant for a device based on the capabilities specified in a configuration.

Chapter 24 describes card files that can specify properties or required files for multiple test classes stored in different directories.

CHAPTER 12

Checking Card Files

In Java Device Test Suite releases prior to version 2.1, card files were authored directly by the test developer. Beginning with version 2.1, card files are (by default) generated automatically from comments you embed in the source code. For more information, refer to Chapter 4.

This chapter describes how to use the Card File Checker tool to verify the required file entries of manually coded or generated card files.

After building the test pack, use the Card File Checker to verify that the required files that appear in the card file corresponds with the required files that appear in the bytecode of the test classes. For information about marking up your Java source files to create a card file, see Chapter 4. For information about directly editing a card file, refer to Chapter 25.

The Card File Checker is not fool proof, however, sometimes it does present you with cause for investigation.

▼ Running the Card File Checker

1. **Change directory to your test pack's `build/` directory.**
2. **Enter one of the following command:**

```
ant run.cardfilechecker
```

The command generates the following standard output:

- A line for each card file including the card file location.
- An indented line for each test in the package including its test class name.
- If a cause is detected, a third indented line appears stating a requirement file is missing (undocumented) or redundant.

FIGURE 12-1 shows Card File Checker output from the example runtime test pack.

FIGURE 12-1 Card File Checker Output Screen - Windows

```
C:\packWorkDir\runtime\build> ant run.cardfilechecker
Buildfile: build.xml

regenerate.packages_list.file:
     [java] Test Suite dir:      .
     [java] Read testsuite structure
     [java] Wrote all necassary files
     [java] Done

set.prop:

check.prop:

common.prepare:
   [delete] Deleting directory C:\packWorkDir\runtime\bin\client\tmpclasses
    [mkdir] Created dir: C:\packWorkDir\runtime\bin\client\tmpclasses

run.cardfilechecker:
     [java] Got params:
     [java]   codebase = bin/client/verified
     [java] [INFO] CardFileChecker started...
     [java] [CARD] com.sun.samples.automated
     [java]   [TEST] SampleAutomatedTest
     [java] [CARD] com.sun.samples.interactive
     [java]   [TEST] SampleInteractiveTest
     [java] [CARD] com.sun.samples.network.client.http
     [java]   [TEST] SampleHttpReadClient
     [java]   [TEST] SampleHttpWriteClient
     [java] [CARD] com.sun.samples.network.client.push
     [java]   [TEST] BasicDynamic
     [java]   [TEST] BasicStatic
     [java]     [redundant] com.sun.samples.network.client.push.PushMIDlet
     [java]   [TEST] RegisterAlarm
     [java] [INFO] CardFileChecker finished. Cards processed: 4. Tests processed
: 7. Problems reported: 1

BUILD SUCCESSFUL
Total time: 4 seconds
C:\packWorkDir\runtime\build>_
```

Note – A test can execute properly with a redundant requirement. However, a missing requirement can cause a test to fail with unpredictable results.

CHAPTER 13

Updating a Test Pack

This chapter describes how to update a test pack. Test packs are usually updated as part of a test pack development cyclical workflow. The most common reasons to update a test pack are as follows:

- Add a test
- Delete a Test
- Fix a bug
- Rename a test
- Rename a test pack property

This appendix is divided into the following sections:

Test Pack Identifiers

When you update a test pack, observe the following rules for changing test pack identifiers in the `testsuite.info` file.

- Be sure *not* to change the value of `TestSuiteID`. The test pack installer uses this value to determine which test pack to replace.
- If you change `TestSuiteName`, the installer installs the update into a new directory as a new test pack and marks the test pack in the old directory that has the same `TestSuiteID`. When a user subsequently launches the harness, the

harness warns that the marked test pack should be uninstalled. If you do not change `TestSuiteName`, the installer replaces the installed test pack of the same `TestSuiteID` and `TestSuiteName`.

- Changing `TestPackName` does not affect test pack installation.

Test Pack Version Identifier

When you begin to revise a test pack, prefix the version number with the characters ++ as described in "Optional Properties" on page 56 (version number is specified in `testsuite.info`) or "Test Pack Versioning Alternative" on page 109 (version number is in `testpack.version.properties`).

When you are ready to deploy the test pack, remove the ++ characters and increment the version number as described in "Optional Properties" on page 56 (version number is specified in `testsuite.info`) or "Test Pack Versioning Alternative" on page 109 (version number is in `testpack.version.properties`)

Test Rename File

If you rename one or more tests in a test pack, you must include a test rename file in the revised test pack. The rename files are `.properties` files. For a complete syntax description of the file, see `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html#load(java.io.InputStream)`.

A rename file maps old test names to new test names. The harness uses the rename file to contend with old test names (created when the name was current) that are found in Java Device Test Suite version 2.0 templates or in Java Device Test Suite version 1.4 profiles.

Each revision of a test pack that includes renamed tests must have its own rename file. Create rename files in the *packWorkDir*`/`*testpackName*`/renames/` directory. A rename file name has the following syntax:

`jdts_testnames_`*oldVersion*`_`*newVersion*`.list`

An example of a rename file name can be `jdts_testnames_1.1_1.2.list`. The identifier `packVersion1.1` is the *oldVersion* and the identifier `packVersion1.2` is the *newVersion*. The version identifiers are from the `TestSuiteVersion` property.

A rename file is a text file containing one line per renamed test. Each line has the following syntax:

fullyQualifiedOldName=fullyQualifiedNewName

An example of a renamed test line is `com.myco.mypk.funct.read.read1=com.myco.mypk.funct.readFile.read1`. In this example the *fullyQualifiedOldName* is `com.myco.mypk.funct.read.read1` and the *fullyQualifiedNewName* is `com.myco.mypk.funct.readFile.read1`.

Test Pack Property Rename File

You can rename test pack properties (those defined in `testsuite.info`), but you cannot rename test class or test case properties. When you rename a test pack property, you introduce a new name for the property and associate it with the test pack version you are creating. The property is not renamed in templates that were created with earlier versions of the test pack.

The harness uses a rename file that you supply to map the alternative names and versions. For example, consider the following scenario of two test pack updates:

- In the first version of a test pack, you define the property Alpha and give it a default value of 1.
- In the second version, you rename Alpha to Beta.
- In the third version, you change Beta's default value from 1 to 2.

Properties in templates include the name and version of the test pack for which they were created. When an administrator installs version 3 of the test pack, the harness finds templates that name the test pack and version 1 and changes the default value of Alpha 1 to 2. It finds templates that name the test pack and version 2 and changes the default value of Beta from 1 to 2. Templates created after the installation of version 3 use the property name Beta and the default value 2. In addition, a version 2 or version 3 test that retrieves the value of Beta when running with a template that defines Alpha, receives the value of Alpha.

To rename a test pack property, follow these steps:

1. **If it does not exist, create a text file named** *packWorkDir*`/renames/propertiesrenamings.properties`**.**

2. **Add one line to the file for each renamed property, using this format:**

   {*TestPackVersion*}*OldPropertyName*=*NewPropertyName*

   TestPackVersion is the value of `TestSuiteVersion` that you give the test pack when you deploy it as described in "Test Pack Version Identifier" on page 90. It is the version number in which the new name is first effective.

   For example, suppose the currently deployed version of the test pack is 1.8 and you are creating version 1.8.1 . The following example renames a property beginning with test pack version 1.8.1:

   ```
   {1.8.1}ImageCropWidth=ImageCropWidthNew
   ```

   In the version 1.8.1 `testsuite.info`, be sure that `ImageCropWidth` does not exist and that `ImageCropWidthNew` does exist.

CHAPTER 14

Reconfigure Environment Settings for Special Test Pack Installation

You can reconfigure environment settings that were set during Java Device Test Suite installation to point the developer's kit to a different Developer Installation.

▼ Reconfiguring the Environment

1. **Open** *installDir*`/admin/shared/lib/devkit/conf/system.properties`**.**
2. **Change the property value of** `JdtsServer.hostname` **to** *host*`:`*port***, where** *host*`:`*port* **is the application server address as seen by the harness.**

   This creates `JdtsServer.hostname=`*host*`:`*port*.

   The port default value is 8080. An example of a `JdtsServer.hostname` value is `JdtsServer.hostname=111.222.333.444:8080`.
3. **Save and close the file.**

CHAPTER 15

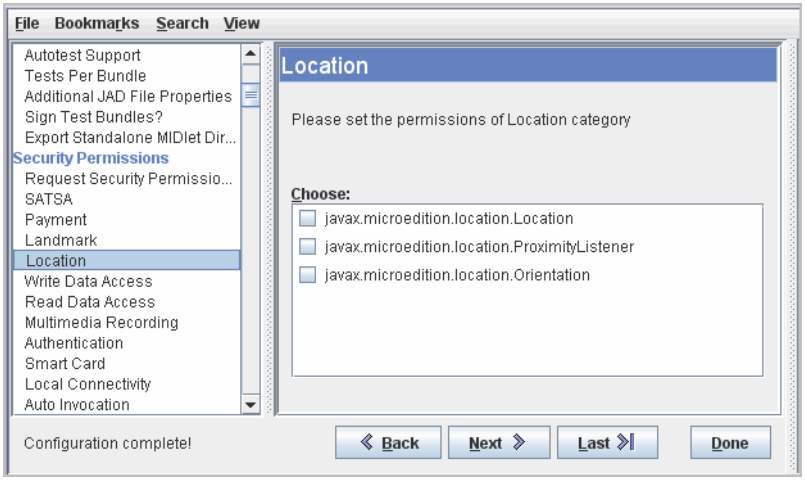
Defining New Security Permissions

Runtime and benchmark tests that use protected APIs (as defined by the MIDP 2.0 security model) must obtain permission to use those APIs. Administrators or testers select the permissions in templates or configurations and the harness adds the permission requests to test bundles.

Note – For OTA test packs, you code permission requests for a MIDlet in its JAD file and manifest file.

FIGURE 15-1 shows how the template and configuration editors display permission categories (groups) in the left pane and, in the right pane, a check box for each permission in the category. In this example, three permissions are associated with the category Location. Test pack documentation tells users what permissions to select.

FIGURE 15-1 Example Permissions



The Security Permissions section of a template or configuration lists the permissions that are known to the Java Device Test Suite. If your runtime or benchmark test pack requires unknown permissions, you define the new permissions in two files in your test pack's root directory (the directory containing `testsuite.info`).

- `policy.txt` defines the permission names and their category.
- `permissions.properties` defines text to be displayed for the category. Technically, this file is optional, but the default values supplied in its absence are not user friendly.

Note – Defining new permissions as described in this chapter is distinct from specifying the permissions a test pack needs in the `testsuite.info` `TSPermissions` property.

When an administrator installs your test pack, your permissions appear in new templates and configurations.

CODE EXAMPLE 15-1 shows the `policy.txt` file format.

CODE EXAMPLE 15-1 Format of `policy.txt` File

```
# This is a comment line
alias: permissionGroup
permission1,
...
permissionN
```

Lines that begin with a pound sign (`#`) are interpreted as comments. *permissionGroup* gives the permissions an identifier that entries in `permissions.properties` can reference. *permission1* through *permissionN* are fully qualified permission names that are to be displayed in *permissionGroup*'s page.

CODE EXAMPLE 15-2 show a `policy.txt` file that defines the permissions shown in FIGURE 15-1.

CODE EXAMPLE 15-2 Example `policy.txt` File

```
alias: LocationPermissions
javax.microedition.location.Location,
javax.microedition.location.ProximityListener
javax.microedition.location.Orientation
```

To specify a category name and descriptive text for a set of permissions, create a `permissions.properties` file and in it name the corresponding *permissionGroup* in a `smry` and `text` entry. Continuing with the example shown in CODE EXAMPLE 15-2, CODE EXAMPLE 15-3 shows the `smry` and `text` items that are displayed in FIGURE 15-1.

CODE EXAMPLE 15-3 Example `permissions.properties` File

```
LocationPermissions.smry=Location
LocationPermissions.text=Please set the permissions of Location category .
```

CHAPTER 16

Properties and Parameter Expansion

This chapter describes properties in an abstract way, independent of the notation used to specify them in source file comments (see Chapter 4), card files (see Chapter 25), or a test pack's `testsuite.info` file (see Chapter 6).

It also describes parameter expansion, which is similar to the like-named mechanism provided by shell programming languages such as BASH. Parameter expansion can be used with property definitions. `@card.requires` comments, and, in OTA test packs, with strings in HTML, WML, and JAD files. Briefly, a string whose value is coded as `${NAME}` is replaced by the run-time value of the property `NAME`.

This chapter has these sections:

Precedence

If you define the same property name in a source file or card file or `testsuite.info` file, the property takes its default value from the most local definition. Thus, if defined in a case and a class, the value is taken from the definition in test case, not from the test class. If defined in a class and in `testsuite.info`, the value is taken from the class. In other words, a property value comes from `testsuite.info` only if it is overridden nowhere else. A property value specified by a user in a template, configuration, or the test tree's Configure Test pop-up overrides the default value.

CODE EXAMPLE 16-1 is a fragment that shows the property `MaxDistance` defined in a class and a case of that class.

CODE EXAMPLE 16-1 Multiply Defined Property

```
 * @testclass
 * @card.property MaxDistance=10
 ...

 * @testcase
 * @card.property MaxDistance=100
 ...

 * @testcase
 * (MaxDistance not defined)
```

In this example, assuming that the tester has not set the value of `MaxDistance`, the first test case has the value 100 because its default has precedence over the class default. The second test case has the value 10 from the test class's definition of `MaxDistance`.

Parameter Expansion

You can code default property values symbolically by naming other properties. At runtime, the system expands the symbolic values into actual values. CODE EXAMPLE 16-2 shows the expansion of the value of `${MaxDistance}`.

CODE EXAMPLE 16-2 Parameter Expansion Example

```
 * @testclass
 * @card.property MaxDistance=10
 ...

 * @testcase
 * @card.property anotherDistance=${MaxDistance}
 * AnotherDistance value is 10 from class definition of MaxDistance
```

Note – Symbolic property values displayed to testers and administrators are not expanded. Continuing with CODE EXAMPLE 16-2, the default value displayed in the Configure Test pop-up window is `${MaxDistance}`.

In an OTA test pack, you use the `${...}` notation to can define parameters in the HTML, WML, and JAD files associated with test MIDlets. To enable expansion of such parameters, you must enable the OTA version of property expansion by setting the `PropertyExpansionProcessing` property to `2.0`. To enable it for a test pack, add the following lines to the `testsuite.info` file:

```
PropertyExpansionProcessing=2.0
PropertyExpansionProcessing.scope=hidden
```

You can also set `PropertyExpansionProcessing` in a source or card file. It is an ordinary property subject to the OTA precedence rules. Accordingly, you can enable or disable expansion class by class or case by case.

To see the effect of `PropertyExpansionProcessing`, consider this example:

- The property myURL has the value `http://theRelay/a.b.c`
- A line in an HTML file is coded
  `... the URL is ${myURL}`
- If `PropertyExpansionProcessing=2.0`, at run time the HTML line is
  `... the URL is http://theRelay/a.b.c`
- If `PropertyExpansionProcessing` is not `2.0`, the HTML line at run time is
  `... the URL is ${myURL}`

Multi-level expansion is supported, that is, nested `${...}` values are permitted. For example, if the property `ac` has the value `x` and `b=c`, then the value `${a${b}}` is expanded to `x`.

For a more realistic example of OTA parameter expansion, see the test case `t06multilayerPropertyExpansion` in *devKitHome*`/tests/ota/src/client/com/sun/samples.interactive/InteractiveJava`. In this example, `PropertyExpansionProcessing` is set to `2.0` in the `testsuite.info` file.

Predefined Parameters for OTA Test Packs

For OTA test packs, you can use the following predefined parameters to set the values of properties in source, card, and `testsuite.info` files. You can use them to set the values of parameters in HTML, WML, and JAD files if `PropertyExpansionProcessing=2.0`.

- `${URL_MIDLET_INSTALL_NOTIFY}` – Expands to the URL of the servlet waiting for installation notification request.

- `${URL_MIDLET_DELETE_NOTIFY}` – Expands to the URL of the servlet waiting for deletion notification request.
- `${URL_STATUS_REPORT}` – Expands to the URL of the servlet expecting the test result report.
- `${URL_JAR_`*NN*`}` – Expands to the URL of the resource defined by the `JARPath`*NN* test case property.
- `${URL_JAD_`*NN*`}` – Expands to the URL of the resource defined by `JADPath`*NN* test case property.
- `${URL_TEST}` – Expands to the full primary URL of the test case that displays in the interactive test description window.
- `${OTA_EXEC_MODE}` – Expands to OTA execution mode (possible values are `html`, `wml`, and `jad`).

Do not define a property that has the same name as a predefined parameter.

CHAPTER 17

Customizing the Test Pack Zip File

The `pack` build target creates an installable zip file from a test pack. The *packWorkDir*`/build/testpack.archive.properties` file controls the zip file generation. You can edit this file to:

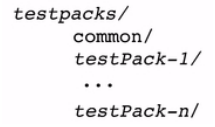
- Exclude files, such as artifacts of your source code control system
- Include additional files, such as a license file
- Change the name of the zip file

CHAPTER 18

Multiple Test Packs in a Directory

For simplicity, this guide states that test pack implementation directories have a single subdirectory called `runtime/`, `benchmark/`, or `ota/`. This convention restricts a work directory to a single test pack of a given type. However, an implementation directory can contain multiple test packs as FIGURE 18-1 shows.

FIGURE 18-1 Generic Work Directory Structure



CHAPTER 19

Using `TestPackInstallerMain` for Faster Test Installation Cycles

A script (called the `tpim` script) is provided for both Windows and Solaris operating systems to invoke the class `com.sun.jdts.tpinstaller.TestPackInstallerMain`:

- *devKitHome*`/tpim.cmd`
- *devKitHome*`/tpim.sh`.

The `tpim` script is a utility shell script to be used as a command-line tool to perform advanced installation tasks such as installing or uninstalling only the server part of a test pack. To see a usage message with a list of supported commands and options, run `tpim` without parameters.

The following example (for the Solaris operating system) demonstrates how to install or update the server part of the sample runtime test pack with a `tpim` script.

```
% cd devKitHome
% ./tpim.sh -install.testpack.dev ./tests/runtime/
```

Customizing the `tpim` Script

The standard `tpim` script that is packaged with the Developer's Kit might need modification to run with your environment. For example, if you upgrade your version of the Java Runtime Environment, you need to edit the script file and change the line that defines the variable `JAVA_EXEC`:

```
JAVA_EXEC="C:\Program Files\Java\jre1.6.0_05\bin\java.exe"
```

CHAPTER 20

Test Pack Versioning Alternative

You must specify a test pack's version in either the `testsuite.info` `TestSuiteVersion` property (see "Optional Properties" on page 56) or in the `TestSuiteVersion` property in `testpack.version.properties`, located in the same directory as `testsuite.info`. The sample and production test packs specify their versions in `testpack.version.properties` files.

Note – If you choose the `testpack.version.properties` alternative, do *not* specify `TestSuiteVersion` in `testsuite.info` because it takes precedence.

Here is an example entry in `testpack.version.properties`:

```
TestSuiteVersion=1.4.1
```

Form the version number from digits separated by periods, following your organization's conventions.

While you are developing or modifying a test pack, prefix the version number with the characters `++`, for example:

```
TestSuiteVersion=++1.4.1
```

These characters ensure that when you launch the harness, it synchronizes templates with the latest changes you have made in the test pack, for example, new properties. When you have finished development, remove the `++` characters, and increment the `TestSuiteVersion` value in one of the following ways:

- Manually with a text editor
- By running `ant inc.testpack.version`, which adds 1 to the low-order digit

After incrementing the version number, run `ant pack` to create the installable test pack.

CHAPTER 21

Build Targets

TABLE 21-1 lists the Developer's Kit Ant build targets.

TABLE 21-1 Build Targets

Target	Description
`clean`	Deletes previously generated files.
`all`	Generates all files according to options specified in `build.properties`. Does not install.
`pack`	Creates an installable zip file of the test pack.
`install`	Invokes `pack`, installs test pack components in Developer Installation and Relay.
`install.srv`	Installs only Relay (server) components (to save time).
`inc.testpack.version`	Increments the `TestSuiteVersion` value in `testpack.version.properties`. Build with this target, or increment the `TestSuiteVersion` value manually when you have finished development. **Note -** This target does not increment `TestSuiteVersion` in `testsuite.info`.
`run.cardfilechecker`	Checks validity of required file statements in manually coded and generated card files.

CHAPTER 22

Tests and Device Features

This chapter describes the optional feature definition file which, if present in your test pack, gives users a way to select tests according to their relevance to features present on a device. To benefit from this chapter, you must understand XML and the XMLSchema language. The chapter covers these topics:

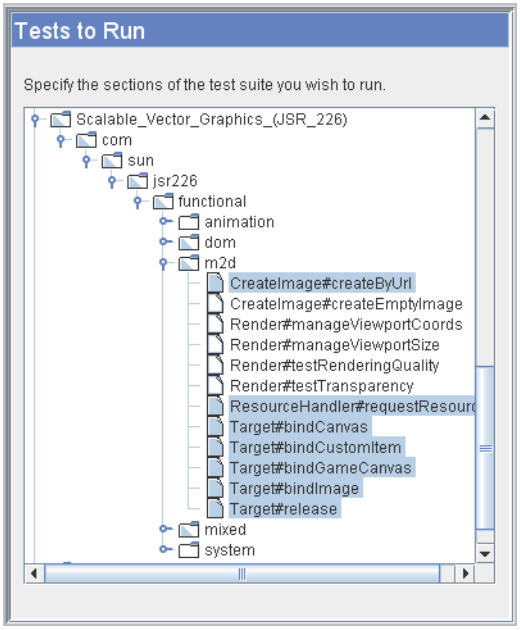
Package and Feature Concepts

A test pack's tests have a package structure and optionally have a feature structure. Java Device Test Suite users can can use these structures to select tests and inspect results in reports.

Package-based Selection and Reporting

FIGURE 22-1 shows part of a configuration's package tree, which users can use to select tests. This tree is called the package tree, though it also contains classes and test cases. The test tree in the main harness window is also a package-based tree.

FIGURE 22-1 Example Package Tree



The standard test report is also package-based, though the package names are "flattened" into long strings. This report is organized in a hierarchy whose top levels are shown in FIGURE 22-2, FIGURE 22-3, and FIGURE 22-4.

FIGURE 22-2 Standard Report Summary

FIGURE 22-3 Standard Report Passed Tests

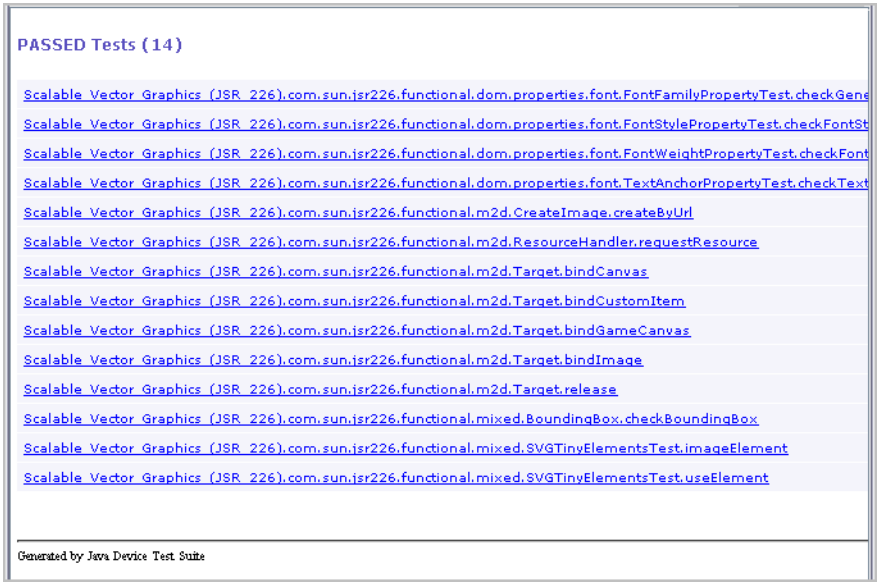
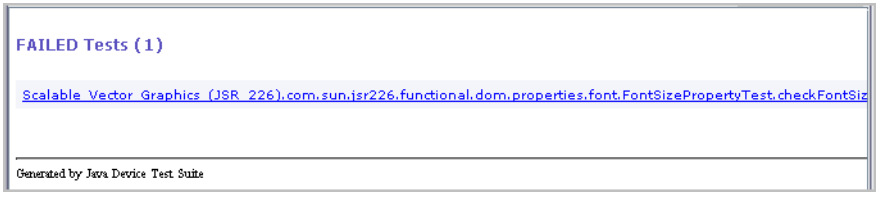


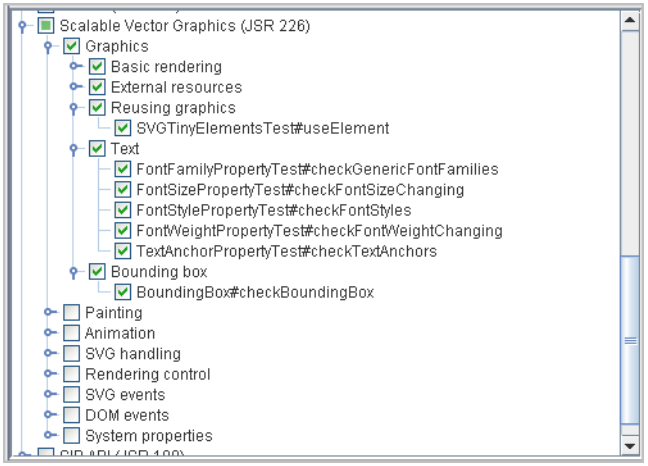
FIGURE 22-4 Standard Report Failed Tests



Feature-based Selection and Reporting

The alternative selection mechanism is feature, sub-feature, and test name using the feature tree shown in FIGURE 22-5.

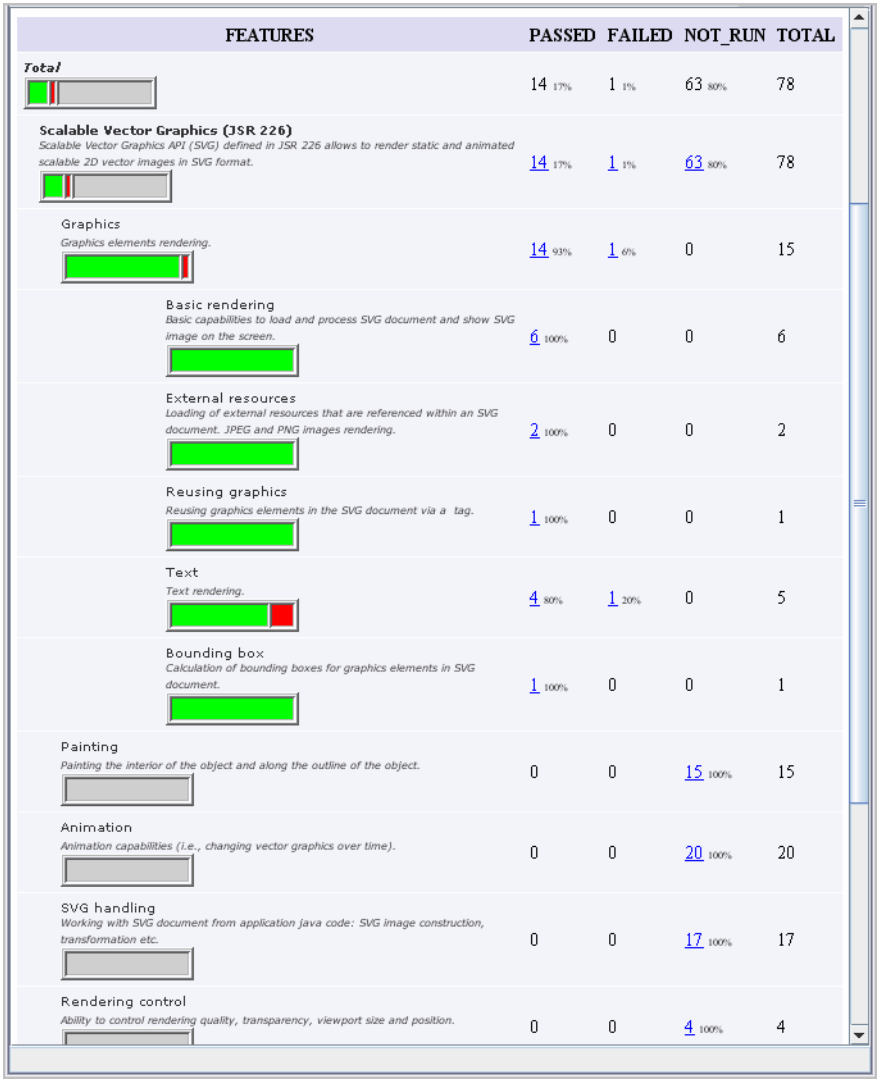
FIGURE 22-5 Example Feature Tree Display



Both the package tree (FIGURE 22-1) and the feature tree (FIGURE 22-5) contain the same tests, but the tests are organized differently. The package tree is more of a developer's structure. The feature tree is more of a user's structure.

When a test pack has a feature definition file, test results can be reported by feature as shown in FIGURE 22-6. The feature-based report makes it easy to see what features have problems and which features have not been tested.

FIGURE 22-6 Feature-based Report



FEATURES	PASSED	FAILED	NOT_RUN	TOTAL
Total	14 17%	1 1%	63 80%	78
Scalable Vector Graphics (JSR 226) *Scalable Vector Graphics API (SVG) defined in JSR 226 allows to render static and animated scalable 2D vector images in SVG format.*	14 17%	1 1%	63 80%	78
Graphics *Graphics elements rendering.*	14 93%	1 6%	0	15
Basic rendering *Basic capabilities to load and process SVG document and show SVG image on the screen.*	6 100%	0	0	6
External resources *Loading of external resources that are referenced within an SVG document. JPEG and PNG images rendering.*	2 100%	0	0	2
Reusing graphics *Reusing graphics elements in the SVG document via a tag.*	1 100%	0	0	1
Text *Text rendering.*	4 80%	1 20%	0	5
Bounding box *Calculation of bounding boxes for graphics elements in SVG document.*	1 100%	0	0	1
Painting *Painting the interior of the object and along the outline of the object.*	0	0	15 100%	15
Animation *Animation capabilities (i.e., changing vector graphics over time).*	0	0	20 100%	20
SVG handling *Working with SVG document from application java code: SVG image construction, transformation etc.*	0	0	17 100%	17
Rendering control *Ability to control rendering quality, transparency, viewport size and position.*	0	0	4 100%	4

For exercises demonstrating both selection and reporting options, see the *Java Device Test Suite Tester's Guide*.

Package and Feature Implementation

When your test pack is installed, it is added to the package tree. If your test pack has the optional feature definition file, that file's contents are added to the feature tree and test results appear in feature-based reports.

Package Design

You can design a test pack's package and class structure for your own convenience. For example, your package structure can model the structure of the specification you are writing tests for. The package structure is expressed in your test pack's directories and class files.

Feature Design

What constitutes a feature (and its optional sub-features) is your decision. When defining features, adopt a test user's (administrator's or tester's) point of view:

- Help the user assess the consequence of a test failure, that is, what area of device functionality (what feature) is compromised by the failure of a particular test. For example, FIGURE 22-6 shows that the test device has a problem displaying text.
- Help the user decide what tests are pertinent for a given device, that is, which tests to run. Consider the functional clusters that different devices, or models of the same device family, are likely to implement. For example, a portion of a multimedia feature tree might look like this:

  Audio Playback MP3

  Via File Connection

  Via HTTP

  Via Input Stream

  Via RTSP

  Audio Playback WAV

  ...

  Video Capture

  ...

  If a test device does not play WAV files, for example, this structure makes it easy to unselect the WAV tests.

Feature Definition File

You define features and their associated tests in a feature definition file named *packWorkDir*`/metadata/features.xml`. The Developer's Kit sample tests include examples.

CODE EXAMPLE 22-1 shows a hypothetical feature definition file that defines the following hierarchy of features:

Bluetooth

- SPP
  - Receive Attribute
  - Remove Service
- L2CAP
  - Receive Attribute
  - Remove Service

In this example, the features happen to correspond closely to the test pack's package structure, but that is a coincidence. A feature can be composed of any set of tests.

CODE EXAMPLE 22-1 Example Feature Definition File

```
<?xml version="1.0" encoding="UTF-8"?>
<features xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="testpack_features.xsd">
    <feature-hierarchy >
        <feature name="Bluetooth">
            <description>Bluetooth communication capabilities</description>
            <feature name="SPP">
                <description>Serial port profile (SPP)</description>
                <feature name="Receive Attribute">
                  <description>Receiving attribute on client side</description>
                    <test name=
"com.sun.jsr082.bluetooth.functional.connection.SPP.receiveInitAttribute" />
                </feature>
                <feature name="Remove Service">
                    <description>Removing service from SDDB</description>
                    <test name=
"com.sun.jsr082.bluetooth.functional.connection.SPP.removeService" />
                </feature>
            </feature>
            <feature name="L2CAP">
                <description>Logical Link Control and Adaptation
Protocol</description>
                <feature name="Receive Attribute">
                  <description>Receiving attribute on client side</description>
```

CODE EXAMPLE 22-1 Example Feature Definition File

```
                    <test name=
"com.sun.jsr082.bluetooth.functional.connection.L2CAP.receiveInitAttribute" />
                </feature>
                <feature name="Remove Service">
                    <description>Removing service from SDDB</description>
                    <test name=
"com.sun.jsr082.bluetooth.functional.connection.L2CAP.removeService" />
                </feature>
            </feature>
        </feature>
    </feature-hierarchy>
</features>
```

A feature definition file must conform to the XML schema defined by *devKitHome*/tests/common/lib/testpack_features.xsd. You can use this schema file to validate your feature definition file in an XML editor. When you build a test pack, the build script validates your feature definition file against the schema and verifies that every test mentioned in a feature definition exists and that every test is mapped to one feature.

CHAPTER 23

Relevance Filtering

If a harness's Preferences specifies that relevance filtering is enabled, when the tester closes the configuration editor, the harness calls an optional test pack method for each test case. This method inspects the configuration and returns null if the test is relevant or a list of reasons describing why the test is irrelevant. If you do not implement the method, all tests in your test pack are relevant.

An irrelevant test is one that is pointless to run for either of two reasons:

- One or more properties in the configuration, which essentially represents the capabilities of the test device, makes a test irrelevant. For example, a property representing the presence of an optional device capability is set to false.
- The test uses an API that requires a permission that the configuration does not request.

The test pack developer writes the relevance criteria code for each test, or accepts the default implementation, which checks configuration permissions. As with other configuration filters, such as keywords and status, the harness does not run a test that fails to "pass" its relevance filter, potentially reducing the time required to execute a test run.

To implement relevance filtering, you extend the abstract class `TestDependencyProvider`. Consult the test-api Javadoc tool documentation for details. For examples, see any Sun test pack, such as *installDir*`/admin/shared/testpacks/Location_(JSR_179)_TestSuite/src/dependencyprovider/com/sun/jdts/lapi/dependencyprovider/LapiDependencyProvider.java`. The key `TestDependencyProvider` method is: `isRelevant(TestId)`:. The harness calls this method for each test case.

There are example relevance filter implementations in *devKitInstallDir*`/tests/runtime/src/client/com/sun/samples/automated/DependencyDemo/`.

CHAPTER 24

Classless Card Files

To define test-related data, such as properties and required files, for the test classes defined in one directory, you use the special source file comments described in "Test Class and Case Comment Blocks" on page 29. To make such test-related information applicable to a collection of classes whose `.java` files reside in different directories, you can define the information in a card file located in a common parent directory that directly contains no classes. Declaring data that applies to a family of classes in a card file minimizes coding and can simplify maintenance, for example, when the name of a required file changes.

Note – Although you can also define card files at the class level, source file comments are recommended instead. See "Writing Card Files Manually" on page 129 for details.
Do not accidentally create a classless card file in a directory that contains test classes. By default, your card file is replaced by one generated from the source file comments.

Topics in this chapter:

Naming, Scope, and Syntax

Card files are optional. When present, with one exception, a card file must be named *package*`.card`, where *package* is the name of the package associated with the same directory. For example, consider the package `com.hypotheticaltestco.uitests`. The card file in the directory named *packWorkDir*`/src/client/com/sometestco/uitests/` must be named `uitests.card`.

The exception is a card file in a directory associated with no package, such as *packWorkDir*`/src/client/com/sometestco/`. Such a card file must be named `default.card`.

The definitions in a card file apply recursively to classes defined in all subdirectories, unless overridden by a definition in a subordinate directory's card file or a comment in a class source file. See "Precedence" on page 99 for details.

In general syntax, a card file is a Java programming language properties file. For a technical description of property file syntax, see
`http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html#load(java.io.InputStream)`

However, there are two differences:

- A card file property can have multiple values. For example, the following multiple assignment is meaningful in a card file:

  ```
  .requires=/x
  .requires=/y
  ```

  In a card file, `.requires` has two values, `/x` and `/y`.

- In a card file, the `\n` escape sequence is not supported in a property value. `\n` can be specified in other card file entries, such as a `doc` attribute that you want to display on multiple lines.

Properties

You can define a property in a card file with the following syntax:

`.property` *PropertyName=DefaultValue*

For example:

```
.property MaxDistance=12
```

You can add the property attributes `scope`, `readonly`, and `doc` as described in "@card.property" on page 40.

Attributes

You can specify the values used to compute the default severity of test failures as in the following examples:

```
.attribute functionality=value
.attribute impact=value
```

See "@card.attribute" on page 43 for details.

Keywords

You can specify keywords that testers can use to filter (subset for execution) tests in subdirectories. The syntax is:

`.keyword=`*keywordList*

keywordList is a space-separated list of keywords. For example:

```
.keyword=interactive security
```

To see the current list of keywords and their definitions, launch the harness and create or open a work directory. Choose Configure > Edit Configuration or Configure > New Configuration. In the Test Selection section of the interview, answer Yes to Specify Keywords? The More Info pane displays the current list of keywords.

Note – The precedence rules do not apply to keyword definitions. For a given test case, the keyword definitions in its source file and applicable (higher-level) card files are cumulative. The test case is tagged with all such keywords.

Required Files

If tests in different subdirectories require files to run, such as media samples or libraries, you can define them in an common ancestor card file. Specify ordinary required files similar to this example:

```
.requires=resources/my_resource1
.requires=resources/my_resource2
```

By the rules in "Naming, Scope, and Syntax" on page 124, this example is logically equivalent to the following, which cannot be expressed in a standard properties file:

```
.requires=/my_resource1,my_resource2
```

Path names in `.requires` entries are relative to *packWorkDir*.

For required zip or JAR files, use `.ziprequires`, for example:

```
.ziprequires=lib/my-lib.zip
.ziprequires=lib/my-lib2.jar
```

Note – The precedence rules do not apply to required file definitions. For a given test case, the required file definitions in its source file and in the card files of higher level directories are cumulative.

PART IV Legacy

This part will help you maintain tests developed in earlier versions of the Java Device Test Suite. Due to the evolution of this product, a number of functions have changed. Some legacy test pack projects might require the use of earlier procedures or tools, which are maintained in the product for backwards compatibility.

Chapter 25 describes how to directly edit a card file rather than having the build system automatically generate the card file.

Chapter 26 describes how to write a `packages.list` file.

Chapter 27 describes how to directly edit an evaluation file rather than having the build system automatically generate it.

Chapter 28 describes writing conditional logging output.

CHAPTER 25

Writing Card Files Manually

In Java Device Test Suite releases prior to version 2.1, card files were authored directly by the test developer. Beginning with version 2.1, card files are, by default, generated automatically from markup you embed in comment blocks in the source code. For more information, refer to Chapter 4.

Note – If you create card files manually in a directory that contains test classes, you *must* add this line to your test pack's `build.properties` file:
`generate.card.files=false`
If you fail to override the default value of `true`, automatically generated card files overwrite the ones you create manually. If you write card files only in directories that have no test classes, as described in "Classless Card Files" on page 123, your card files are preserved regardless of the value of `generate.card.files`.

A tool exists for checking the contents and consistency of card files, regardless of whether they were generated or authored directly. For more information, refer to Chapter 12.

This chapter describes how to directly edit a card file. It contains the following sections:

While the `testsuite.info` file describes a test pack as a whole, card files describe the contents of directories that contain test classes. The word "card" has no special significance. Every directory that contains one or more test classes must have a card file that names the test classes, the test cases they contain, related files they access, and properties they use.

The card file naming structure is *containingDir*`.card`, where *containingDir* is the unqualified name of the directory containing the card file. For example, a directory named `uitests` must have the card file `uitests.card`.

In general syntax, a card file is a Java programming language properties file. For a technical description of property file syntax, see
`http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html#load(java.io.InputStream)`

However, there are two differences:

- A card file property can have multiple values. For example, the following multiple assignment is meaningful in a card file:

  ```
  Class.requires=/x
  Class.requires=/y
  ```

  In a card file, `Class.requires` has two values, `/x` and `/y`.

- In a card file, the `\n` escape sequence is not supported for property values.

You can write card file lines in any order. However, be sure to observe the syntax and requirements noted in the following sections. The harness does not check for invalid entries. An error can cause unexpected behavior by the harness or a test.

CODE EXAMPLE 25-1 shows a simple card file for a directory containing two test classes, `SampleAutomatedTest1.java` and `SampleAutomatedTest2.java`, and a class used by `SampleAutomatedTest1.java`, `RequiredClass.java`.

CODE EXAMPLE 25-1 Simple Card File

```
# Specify the test cases in a test class
SampleAutomatedTest1=TestCase1,TestCase2

# Specify a file needed by a test case in this class
SampleAutomatedTest1.requires=RequiredClass.class

# Specify a property used by a test case, and its default value
SampleAutomatedTest1.property=TestProp=test

# Specify the test case in a second test class
SampleAutomatedTest2=TestCase1
```

Comment Lines

A comment line begins with a pound sign (`#`), for example:

```
# This is a comment.
```

The harness ignores comment lines.

Test Case Definitions

A card file must contain one line for each test class in the directory. The line names the test class and its test cases, separated by commas, for example:

 # Test class (SpriteTest.java) with a single test case (Rotate)

```
SpriteTest=Rotate
```

 # Test class with two test cases

```
CollisionTest=SpritePixel,ImageBounds
```

Note – Do *not* separate test case names with space characters. Such a space causes the build to fail.

In this guide, a test class named in a card file is referred to as *TestClassName*, and a test case named in a card file is referred to as *testCaseName*. In the previous examples, `SpriteTest` and `CollisionTest` are *TestClassNames* and `Rotate`, `SpritePixel`, and `ImageBounds` are *testCaseNames*.

Required File Definitions

If a test case requires another file (for example, a helper class or a file containing an image or a sound), name the file (or files) as shown in CODE EXAMPLE 25-2.

CODE EXAMPLE 25-2 Example Required File Definitions

```
SpriteTest.requires=com/sun/midp/images/sprite1.png
SpriteTest.requires=EncErr_Streams.class
```

In this example, `SpriteTest` is the test class name.

Observe the following requirements when writing path names:

- The path name separator must be a slash (/) character.

- If the path name contains a separator (as in the first line in the CODE EXAMPLE 25-2), the path is relative to the directory specified in the test pack property `TestClassesDir`. For example, for runtime test packs, this directory is typically `bin/client/verified` (relative to the test pack root).
- If the path name does not contain a separator (as in the second line in CODE EXAMPLE 25-2), the file is in the same directory as the card file.

Note – Properties are not inherited. In particular, required file properties declared in class `X` do not apply to `X`'s superclass or subclasses. Be sure to declare the files that each class requires.

After a build, you can use the Card File Checker utility to verify that all of the required files are recorded with a `.requires` property. For more information, see Chapter 12.

Property Definitions

If a test class or case needs a user-editable property, name the property and specify its default value with the following syntax:

TestClassName`.property=`*PropertyName*`=`*DefaultValue*

For example:

```
SpriteTest.property=MaxDistance=12
```

To add a user-editable property to a case, use the following syntax:

TestClassName`.`*testCaseName*`.property=`*PropertyName*`=`*DefaultValue*

For example:

```
MaterialBasic.setGetShininess.property=MaxDistance=12
```

You can use the same property name for different test classes or cases that are in the same card file, but if you do so, you are defining different properties. The following example defines two properties that have the same name.

```
SpriteTest.property=MaxDistance=12

ImageTest.property=MaxDistance=20
```

In this example, if `SpriteTest` retrieves `MaxDistance` with `Runner.getProperty()` as described in "Obtaining a Property Value" on page 61, it receives the value 12 (assuming the user has not changed the default value). If `ImageTest` retrieves `MaxDistance`, it receives the value 20.

A user can inspect or override a default test case property value from the tester edition of the harness. To inspect or override a default test case property value, right click the test case in the harness tree and choose Configure Test.

Note – If a test class extends a superclass, and the superclass needs a property that is undefined in `testsuite.info`, specify the property for the test class (the subclass). For example, if test class `X` extends class `Y`, and class `Y` uses property MaxSize, make this card file entry:
`X.property=MaxSize=`*value*.

Class and Case Keyword Definitions

Use a *keyword* property definition to provide the harness with a list of keywords to filter (subset) test cases. Select a keyword based on its descriptive relevance to the test class or test case from the perspective of the tester. To see the current list of keywords and their definitions, launch the harness and create or open a work directory. Open an interview from the harness menu bar by means of Configure > Edit Configuration or Configure > New Configuration. In the Test Selection section of the interview, answer Yes to Specify Keywords? The More Info pane displays the current list of keywords.

Use the `keyword` specifier in the card file to specify the keyword for an individual test case or all of the test cases in a test class. A test case can have more than one keyword. Multiple keywords are separated by spaces.

Use one of the following syntaxes in a card file to add keywords:

- *TestClassName*`.keyword=`*keyword1 keyword2*

  In this case, all test cases under *TestClassName* are marked with the list of mentioned keywords.

- *TestClassName*`.`*testCaseName*`.keyword=`*keyword1 keyword2 keyword3*

  In this case, only individual *TestClassName*`.`*testCaseName* test case are marked with the list of mentioned keywords.

Note – All interactive tests *must* have the keyword `interactive`.

Note – Avoid adding incorrectly named keywords, as this is not checked at any step.

Special Property Definitions

Use a special property definition to direct the harness to add a line to the JAD file that it creates for each test bundle. In effect, a special property definition is a way to pass a static parameter to an application management system (AMS), sometimes called a Java application manager. The static parameter only applies to the test bundle associated with the JAD file.

Use the following syntax in a card file to add a line to the JAD file:

ClassName`.specialproperty=<jad>.`*n*`=`*LineToAdd*

n is a number that distinguishes multiple `specialproperty` entries in the same card file. The harness does not interpret *LineToAdd* but simply copies it to the JAD file. The test device AMS interprets *LineToAdd* when it downloads the JAD file. It is the test developer's responsibility to ensure that *LineToAdd* is meaningful to the AMS and follows the JAD file syntax. The JAD file is defined in the MIDP 1.0 specification.

For example, to add a line to a JAD file associated with a test bundle containing, `MIDlet2:AlarmMidlet` include a line like this in the card file:

```
SpriteTest.specialproperty=<jad>.1=MIDlet2:AlarmMidlet
```

Choosing Between Card File and `testsuite.info` Properties

Define a property for a test case to use in either the `testsuite.info` file or a card file. Consider the following to make your choice:

- If test cases in multiple test classes use a property, it is easier to define it in the `testsuite.info` file, which makes it accessible to all test cases in the test pack.
- If a property must be accessible to administrators, define it in the `testsuite.info` file.
- If a property only needs to be accessible to users, and is only used by one or more test cases in one test class, define the property in the card file.

CHAPTER 26

Writing `packages.list` Files

Each test pack must have a `packages.list` file in its `build/` directory. The `packages.list` file names the directories that contain `package.html` files. By default, the build system creates the `packages.list` file for you. You can alternatively create it with a text editor.

The `packages.list` file is used in these two ways:

- It defines the packages for which the build system generates documentation.
- It defines the packages for which the build system generates `.card` and `.html` files. See Chapter 4 for more information.

To create the `packages.list` file manually, observe the following guidelines:

- The file has the format of a Java programming language properties file.
- The file contains a single property `DOC_PACKAGE`, whose value is a list of space-separated package names.
- In the test pack's `build.properties` file, set `generate.packages.list.file` to `false`.

CHAPTER 27

Writing Evaluation Files

For each test case in an interactive or OTA test class, an evaluation file is required, which instructs the user on manipulating and observing the device and deciding if the test results indicate success. (For an OTA test class, the evaluation file also instructs the user to download a MIDlet to the test device.)

Starting with version 2.1 of Java Device Test Suite, evaluation files are, by default, generated automatically from markup you embed in comment blocks in the source code. Refer to Chapter 4 for details.

Note – If you choose to create evaluation files manually, add this line to your test pack's `build.properties` file:
`generate.evaluation.files=false`

This chapter is intended for legacy situations in which the evaluation files should be directly written.

▼ Procedure for Editing an Evaluation File

1. **Locate the file**
   devKitHome`/tests/runtime/src/client/com/sun/samples/interactive/SampleInteractiveTest.example1.html`**.**
2. **Change the file name to** *TestClassName*`.`*testCaseName*`.html`**.**
   Refer to "Test Case Definitions" on page 131 for the meaning of *TestClassName* and *testCaseName*.
3. **Edit the file as follows:**
   a. **Replace the Sun copyright lines with text that conforms to your organization's copyright policy.**

b. **Enter the test name that you want to display in the window title bar between the title tags (such as `<title>My Interactive Test</title>`)**

c. **Change `SampleInteractiveTest.TestCase1` to** *TestClassName*.*testCaseName*.

Refer to "Test Case Definitions" on page 131 for the meaning of *TestClassName* and *testCaseName*.

d. **Change `Specify the objectives of the testcase` to your test case objectives.**

e. **Change `Instruct the user on any interaction that is required` to your interaction instructions.**

f. **Change `Specify the expected behaviour` to the behavior the device will exhibit.**

g. **Change `Any additional comments` to your comments or a space.**

4. **Save and close the file.**

Example Evaluation File Text

CODE EXAMPLE 27-1 Sample Interactive Test Evaluation File Text

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!--
 @(#)SampleInteractiveTest.TestCase1.html

 Copyright 2007 Sun Microsystems, Inc. All rights reserved.
 Use is subject to license terms.
-->

<html>
<head><title></title></head>
<body>

<table summary="Test Description" border=2>
   <tr>
      <td><b>Test Name</b></td>
      <td>SampleInteractiveTest.TestCase1</td>
   </tr>

   <tr>
      <td><b>Test Objectives</b></td>
```

CODE EXAMPLE 27-1 Sample Interactive Test Evaluation File Text *(Continued)*

```
      <td>Specify the objectives of the testcase.</td>
   </tr>

   <tr>
      <td><b>User Interaction</b></td>
      <td>Instruct the user on any interaction that is required.</td>
   </tr>

   <tr>
      <td><b>Test Expected Result</b></td>
      <td>Specify the expected behavior.</td>
   </tr>

   <tr>
      <td><b>Comments</b></td>
      <td>Any additional comments if needed.</td>
   </tr>
</table>

</body>
</html>
```

▼ Including Reference Images in an Evaluation File

FIGURE 27-2 shows an example of an evaluation window with references images.

1. **Copy the image files to the package directory or a subdirectory, such as** `images/` **or** `jpg/`**.**
2. **Add a Reference Images row to the bottom of the table in** *TestClassName*`.`*testCaseName*`.html`**.**

   CODE EXAMPLE 27-2 shows an example.

CODE EXAMPLE 27-2 Example Reference Image Row

```
  <tr>
    <td><b>Reference Images</b></td>
    <td>
    <img src="./jpg/image1.jpg"><font size="3" >Ref1</font>
    <img src="./jpg/image2.jpg"><font size="3" >Ref2</font>
    </td>
  </tr>
```

3. **Add instructions to the Comments row to help users compare the reference images to the images displayed by test devices.**

   FIGURE 27-2 gives an example.

4. **(Optional) Refer to the images in the User Interaction section of the test evaluation file.**

5. **Ensure that the reference image's file extension appears in the test pack's `common/build/build.properties` file.**

   The default extensions, such as `.html` and `.gif`, are listed after `client.res.pattern`.

   If necessary, override this section in your test pack's `build/build.properties` file.

An Evaluation File Rendered by the Harness

FIGURE 27-1 shows an evaluation window rendered by the harness using the HTML source file from CODE EXAMPLE 27-1.

FIGURE 27-1 Sample Interactive Test Evaluation Window

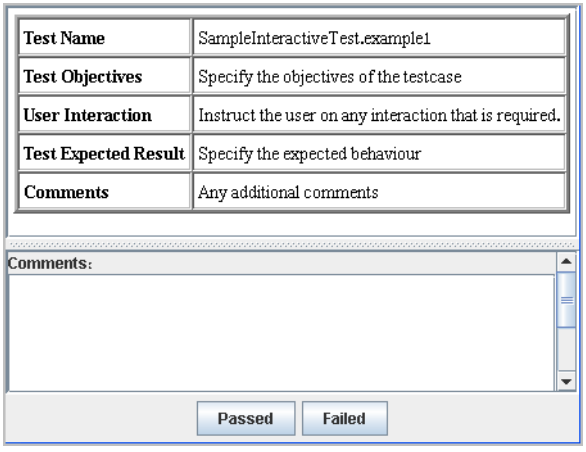
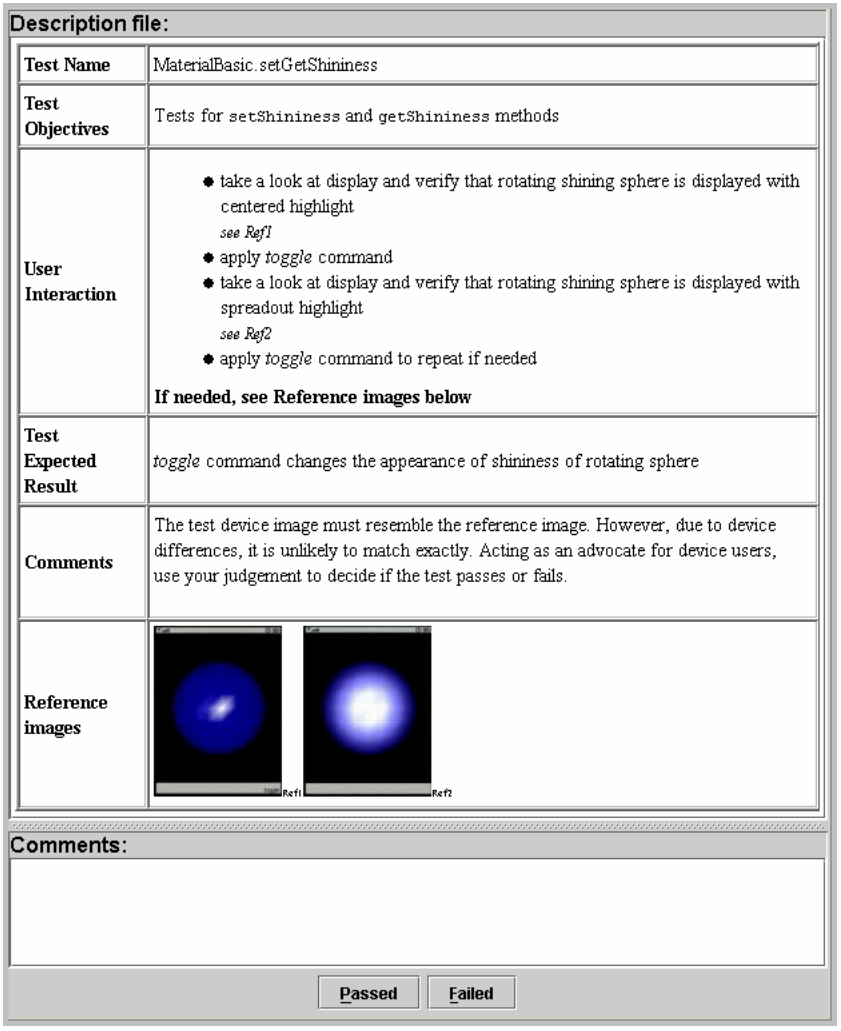


FIGURE 27-2 Example Test Evaluation Window With Reference Images



CHAPTER 28

Writing Conditional Output

New test packs should use the Logger API (see "Logging" on page 62) to write log file output. This chapter describes the legacy logging facility in the context of Java Device Test Suite version 2.2 or later.

The `Runner.verbose()` method conditionally writes a string to the harness test log, depending on the `verboseLevel` argument passed to it and the value of the harness's Run Tests > Log Messages menu item. CODE EXAMPLE 28-1 shows the combinations that result in output.

CODE EXAMPLE 28-1 Conditions for Log Output

Flag Argument	Conditions When Written
`Runner.VERBOSE`	Run Tests > Log Messages > Verbose or lower
`Runner.DEBUG`	Run Tests > Log Messages > Debug or lower

In general, use `Runner.DEBUG` to document normal operations. The exception is messages noting the start of a test case, which uses `Runner.VERBOSE`. Use `Runner.VERBOSE` for test failure messages and messages providing information about failures.

CODE EXAMPLE 28-2 shows an example that produces output if the tester has has set Run Tests > Log Messages to Debug or lower.

CODE EXAMPLE 28-2 Writing a Diagnostic Message

```
Runner.verbose("The enumeration has " + renum.numRecords() +
  " records", Runner.DEBUG);
```

PART V Appendices

This part contains supplementary information about this release.

Appendix A describes how to adapt the WMA test emulator to run Cell Broadcast Short Message Service (CBS) tests.

Appendix B describes the use of exclude lists to prevent specified tests from being run.

Appendix C summarizes the files in the distribution that have changed with this 2.4 release.

APPENDIX **A**

Adapting the WMA Test Emulator

This appendix describes how to adapt the emulator side of the Java Device Test Suite Wireless Messaging API (WMA) tests to your test device. This adaptation is required to run Cell Broadcast Short Message Service (CBS) tests.

Note – This appendix has nothing to do with Java Device Test Suite test development. It describes software that must be developed to run WMA CBS tests. It is included in this guide because writing this code is a developer task.

The appendix consists of these sections:

Test Types

Some of the WMA tests are network tests (see "Writing Network Tests" on page 65). These tests have two parts. The device part exercises WMA interfaces on the test device. The emulator part coordinates with the device-resident part, sending, receiving, and analyzing test messages as needed.

For CBS tests, the emulator part is called a CBS Server. It is called to send a valid CBS message to the test device.

Implementing `CBSServer`

CODE EXAMPLE A-1 shows the `CBSServer` interface.

CODE EXAMPLE A-1 `CBSServer` Interface

```
package com.sun.wma.api.server;

public interface CBSServer {
  public void send (String type, int segNum, String address);
  public void init ();
  public void die ();
}
```

TABLE A-1 specifies the semantics of a `CBSServer` implementation.

TABLE A-1 `CBSServer` Implementation Guide

Method/Parameter	Description
`send()`	Sends a CBS message through the mobile network.
`type`	`"gsm7"` `"ucs2"` or `"binary"` represent three encoding types defined by the WMA specification.
`segNum`	Number of message segments as defined by Appendix A of the WMA specification.
`address`	CBS address for the message. See Appendix B of the WMA specification for examples.
`init()`	CBS server initializes itself, if necessary.
`die()`	CBS server terminates any active operations and clean up resources.

▼ Deploying the Implementation

1. **Compile the implementation.**

   The following shows one simple way to compile. Other ways might also be valid.

   `% javac -d` *DestDir File*

   - *DestDir* is the directory where the compiled class is placed.
   - *File* is the file to compile.

2. **Put the compiled files into a JAR file.**

   For example:

   `% jar cvf` *CBSImplJar CBSImplClass*

   - *CBSImplJar* is the name of the JAR file you are creating.
   - *CBSImplClass* is the name of the class file created in Step 1.

   You can use more than one JAR file.

3. **Use the administrator edition of the harness to create a section containing the WMA test pack.**

   The online help describes how to create and configure a template.

4. **Connect the test device to the harness host.**

   The *Java Device Test Suite Tester's Guide* describes how to connect test devices and how to launch and use the harness.

5. **Launch a harness.**

6. **Open the Configuration Editor and go to the WMA section created in Step 3, and select the WMA test pack.**

7. **Answer the corresponding questions to set the following properties for the CBS Server of the WMA test pack in the template.**

   a. **Set `ServerImplJarLocation` to the absolute path name of the JAR file or files created in Step 2.**

      Where the `ServerImplJarLocation` property value contains more than one JAR file path, the paths must be semicolon separated.

   b. **Set `CBSServerImpl` to the full class name (including package name) of the `CBSServer` implementation class.**

8. **In the harness, open the Test Server Monitor to see diagnostic messages from the WMA test emulator.**

9. **Click the Start button.**

APPENDIX **B**

Exclude Lists

If your test pack has tests that you do not want testers to run, you can ask administrators or testers to name the tests in an exclude list file, and to specify that file in templates or configurations. Testers can see excluded tests but cannot run them.

The relevant interview items are Specify an Exclude List and Specify Exclude List Files. The *Java Device Test Suite Tester's Guide* describes the exclude list file format. You can supply the file yourself, or give the data needed to create the file to administrators or testers.

APPENDIX **C**

Change Log

This appendix describes the main changes to the Developer's Kit and the *Developer's Guide* for this release and recent past releases.

New in Release 2.4

This section summarizes how the 2.4 Developer's Kit and *Developer's Guide* differ from their immediate predecessors.

- `@card.requires` comments in source files can use parameter expansion to specify path name components symbolically. CODE EXAMPLE 4-3 gives an example.
- `@card.requires` can be specified for test cases as well as classes.
- The new `@card.ziprequires` card file entry specifies a required zip or JAR file. See "Required Files" on page 126.
- The server part of a network test can obtain property values. See "Writing the Server Part" on page 66.
- In this guide, an erroneous note that said that `testsuite.info` properties cannot refer symbolically (`${`*propertyName*`}`) to other property values has been removed. The following usage is legal in a `testsuite.info` file:

```
A=1
C=${A}
```

- OTA applications (MIDlets) can send log messages to the harness. See "Application Logging" on page 82 for details.
- Test servers (see "Writing Network Tests" on page 65) that use keystores must be modified due to a change in the interface `ResourceHelper`. See *devKitHome*`/docs/test-server-api/com/sun/midp/testmanager/ResourceHelper.html` for details including an example.

- *devKitHome*`/docs/` contains the Test Design Policies document that Sun engineers use when writing tests.
- In source file comment blocks, lines can be continued and non-ASCII characters can be specified. See "Comment Block Overview" on page 29 for details.

New in Release 2.3

This section summarizes how the 2.3 Developer's Kit and *Developer's Guide* differ from their immediate predecessors.

- The use of `policy.txt` and `permissions.properties` to define new permissions has been clarified. See "Defining New Security Permissions" on page 95.
- Ant 1.7.1 is required.
- A new chapter, "Tests and Device Features" on page 113, describes the optional feature definition file.

New in Release 2.2

This section summarizes how the 2.2 Developer's Kit and *Developer's Guide* differ from their immediate predecessors.

- Samples illustrate the multi-level `Logger` API for generating log messages. A new class, `Level`, has been added to the test API.
- The description of the `scope`, `readonly`, and `doc` attributes that can be specified for test class and case properties has been corrected. See "`@card.property`" on page 40.
- Property expansion has been unified in runtime, benchmark, and OTA tests. All behave as OTA tests did in previous recent releases: A value in a case definition overrides a value in a class definition, and a value in a class definition overrides a value in a `testsuite.info` definition. In other words, if a user does not specify a property value, `Runner.getProperty()` always returns the most local default value. "Properties and Parameter Expansion" on page 99 gives the details.
- Property expansion applies to special properties.
- `filter` is deprecated in property descriptions. Use `keyword` instead.
- `@precondition` is not required in test class or case comment blocks.

- The formerly required `testsuite.info` property `SupportedFrameworks` has been removed from this guide because it is no longer used.
- JDK version 1.5 is no longer supported. Use JDK version 1.6.0_03.
- The chapter "Converting 1.4 Test Suites to 2.0 Test Packs" has been removed because Java Device Test Suite version 1.4 is no longer supported.
- You can rename test pack properties while preserving compatibility with templates that use the old name. See "Test Pack Property Rename File" on page 91 for details.
- Two methods, `close()` and `flush()` have been added to the Logger API.
- The role of test pack identifiers in the `testsuite.info` file, such as TestSuiteName, has been clarified for new and updated test packs. See "Required Properties" on page 55 and "Test Pack Identifiers" on page 89.
- Test classes and cases can be assigned functionality and impact codes, from which the Java Device Test Suite calculates and reports the severity of a test failure. See "`@card.attribute`" on page 43 for details. Sample tests have been updated with these codes.
- The interaction of push test components has been corrected, see "Architecture of a Push Test" on page 67.
- The descriptions of test pack versioning have been corrected and clarified. See "Optional Properties" on page 56 , "Test Pack Versioning Alternative" on page 109, and "Test Pack Version Identifier" on page 90.
- The role of the optional `policy.txt` file for defining new permissions has been clarified. See "Defining New Security Permissions" on page 95.
- The use of the optional `TSPermissions` `testsuite.info` property has been clarified and corrected. See "Optional Properties" on page 56.
- A new appendix "Exclude Lists" on page 151 describes how exclude lists can disable execution of buggy or incomplete tests.

New in Release 2.1.2

No change in this release.

New in Release 2.1.1

This section summarizes how the 2.1.1 Developer's Kit differs from its predecessor.

- Advanced property values can be defined with markup in comment blocks for test cases and cases, not just in the `testsuite.info` file for test packs.
- The conceptual diagrams in Chapter 1 of this manual showing the test types have been updated.
- The version of the test pack can now be incremented by building with the ant target `inc.testpack.version`. For more information, see the property `TestSuiteVersion` under "Required Properties" on page 55.
- Sample test classes in the DevKit now demonstrate advanced property attributes such as `.type`, `.min`, `.max`, `.values`, and `.readonly`. This is now documented in Chapter 4 of the *Developer's Guide*.
- Some environment variables have been renamed:

  `me.home` is now called `client.platform.home`

  `me.lib` is now called `client.platform.lib`
- The string "jtwi" has been eliminated from build files.

New in Release 2.1

This section summarizes how the 2.1 Developer's Kit differs from its predecessor.

- The *Developer's Guide* has been reorganized to separate material that is of interest to particular sets of readers, such as advanced and legacy topics.
- By default, card files and interactive test evaluation files are automatically generated from comments embedded in source files. See Chapter 4.
- By default, the `packages.list` file is generated automatically.
- In the sample distributed test documentation, the terms UUT (unit under test) and UE (execution unit) have been changed to TD (test device) and PD (partner device) for better consistency with product terminology. Test function is not affected.
- The officially supported Java Developer's Kit is version 1.6.0_03. By default, tests are compiled to run with 1.5 JRE software.
- The `$IPFILTER` macro is provided for static push MIDlet registration. See "Using `$IPFILTER`" on page 71.
- The following new sample illustrates the use of log trace levels: *devKitHome*`/tests/runtime/src/client/com/sun/samples/automated/Logging.java`.
- A new chapter in this guide explains property value precedence and parameter expansion. See Chapter 16.

- `${TS_DIR}` refers only to the Relay file system root. Formerly, it could also refer to the test pack root in the Developer Installation.
- You can customize the text displayed for new permissions questions. See Chapter 15 for details.
- You can specify text for configuration and template editors to display as a category name and description. See "Categories" on page 52.
- You can specify the location of server source files in a `testsuite.info` file with the new `TestServerSourcesDir` property. The value was formerly hard coded as `src/server`, which remains the default. The runtime sample includes a usage example.
- The structure of the MIDlet source files in the OTA sample test pack has been changed.
- You can specify the location of MIDlet files with the new `testsuite.info` `MIDletDir` and `MIDletSourcesDir` properties. The sample distributed and OTA test packs have usage examples.
- By default, the build creates a "catalog" of test pack components in the `listings/` directory in the test pack root. You can disable generation by setting `build.test.list=false` in the test pack's `build.properties` file. The `listings.dir` property specifies the output directory.

Index

J

K

M

N

O

P

R

S

T

U

V

Z