Java™ ME TCK Framework Developers Guide

Version 1.2.1

Sun Microsystems, Inc.
www.sun.com

Part No. 08-2008
August 2008

Please

Adobe PostScript

Contents

Figures

Tables

Code Examples

Preface

This guide describes how to use resources from the Java™ Platform, Micro Edition Technology Configuration Kit Framework (Framework) to develop and configure test suites and tests for Java Platform, Micro Edition (Java ME platform) technologies.

Before You Read This Book

To fully use the information in this document, you must read and understand the topics discussed in the following books:

- *TCK Project Planning Guide*

  A high-level planning guide that describes a process for developing a Technology Configuration Kit (TCK). A TCK is a suite of tests, tools, and documentation that enable an implementor of a Java technology specification to determine if the implementation is compliant with the specification. This guide is available from the Java ME Technology APIs and Docs web site at `http://java.sun.com/javame/reference/apis.jsp`.

- *Java Technology Test Suite Development Guide*

  Describes how to design and write tests for any TCK. It also provides "how-to" instructions that describe how to build a TCK and write the tests that become the TCK’s test suite. This guide is available from the Java ME Technology APIs and Docs web site at `http://java.sun.com/javame/reference/apis.jsp`.

- *JavaTest Architect’s Guide*

  This guide provides a description of the process of creating test suites, and configuration interviews that JavaTest™ harness (harness) can run. This guide is available from the Java ME Technology APIs and Docs web site at `http://java.sun.com/javame/reference/apis.jsp`.

- *Graphical User Interface User's Guide*

  This guide provides a description of using the harness Graphical-User Interface (GUI). This guide is available from the Java ME Technology APIs and Docs web site at `http://java.sun.com/javame/reference/apis.jsp`.

- *Command-Line Interface User's Guide*

  This guide provides a description of using the harness command-line interface. This guide is available from the Java ME Technology APIs and Docs web site at `http://java.sun.com/javame/reference/apis.jsp`.

Intended Audience

This guide is intended for Java ME technology test suite developers and test writers who are using the Java ME TCK Framework resources to create test suites.

How This Book Is Organized

Chapter 1 introduces using the Framework resources to develop test suites for the Java ME platform.

Chapter 2 describes the procedure for installing the Framework bundle on a development system.

Chapter 3 describes the process of creating a simple automated test and updating a test suite that uses Java ME TCK Framework resources.

Chapter 4 describes the process required to build test suites that use Framework resources.

Chapter 5 describes the types of Java ME technology tests that can be written.

Chapter 6 describes the Java ME agent used in conjunction with the test harness to run tests.

Appendix A describes the test Application Programming Interfaces (APIs) for different types of test suites.

Appendix B describes the contents of the Java ME TCK Framework bundle.

Appendix C contains a summary of the Framework test description fields and keywords.

Glossary contains the definitions of words and phrases used in this book.

Platform Commands

This document does not contain information about basic platform 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 at http://docs.sun.com

Examples

Examples in this guide might contain the following shell prompts:

Shell	**Prompt**
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Examples in this guide might also contain the ^ character at the end of a line to break a long line of code into two or more lines. Users must type these lines as a single line of code.

Typographic Conventions

This guide uses the following 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. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. To delete a file, type **`rm`** *filename*.

* **The settings on your browser might differ from these settings.**

Related Documentation

When installed, the Framework includes a doc directory that contains both Framework and harness documentation in PDF and HTML format.

The following documentation provides detailed information about the Java programming language and the harness included with this release.

Application	Title
JavaTest Harness	*JavaTest Harness User's Guide: Graphical User Interface* *JavaTest Harness User's Guide: Command-Line Interface* *JavaTest Architect's Guide*
Programming Reference	*The Java Programming Language*
Programming Reference	*The Java Language Specification*

Accessing Sun Documentation Online

The Java Developer Connection™ program web site enables you to access Java platform technical documentation at `http://java.sun.com/`.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Provide feedback to Sun at
`http://java.sun.com/docs/forms/sendusmail.html`

CHAPTER 1

Introduction

This chapter introduces using the Framework resources to develop test suites for the Java ME platform, including descriptions of the kinds of tests that a test suite can include. This chapter contains the following sections:

- Getting Started
- Using the Framework
- Test Types

Getting Started

For test developers who do not already have an understanding of the TCK and test suite development process, it can take considerable time and effort to put the pieces of the TCK puzzle in place. Consequently, test developers should read the *TCK Project Planning Guide* and the *Java Technology Test Suite Development Guide* to understand the principles involved in constructing TCKs and test suites before using this guide. These documents provide the following fundamental information about TCK and test suite development:

- For test developers who are creating a TCK, the *TCK Project Planning Guide* provides general descriptions of the components required for a TCK, explanations of the process of creating a TCK, descriptions of the resources required to create a TCK, and a description of the planning process for a TCK development project.
- For test developers who are creating a test suite, the *Java Technology Test Suite Development Guide* provides general descriptions of the methods used to create a test suite with its tests and include an overview of test development techniques.

  The chapters in this document are presented in a sequence that developers who are new to the process of creating test suites can follow when creating their own test suite.

Note – The *TCK Project Planning Guide* and the *Java Technology Test Suite Development Guide* documents were originally developed as part of a product used by the Java Community Process (JCP) Specification Leads. For that reason, these documents refer to JCP procedures that might not apply to a test developer's Framework project.

After becoming familiar with TCK and test suite components and development processes, test developers can begin using this Developer's Guide to write tests and create Java ME technology test suites that use Framework resources.

To help familiarize test developers new to the Framework with the process of writing tests and building test suites, the chapters in this Developer's Guide are presented in the following sequence that begins with examples of simple tasks (such as writing a simple test and updating a test suite) and progressively introduces the more complex tasks of creating custom test suites, tests, and test configurations:

- Chapter 3 describes how to write a simple test that can be added to an existing test suite (provided by the Framework) and run by the harness.
- Chapter 4 describes how to use Framework resources when constructing a custom test suite and test configuration.
- Chapter 5 describes how to use Framework resources when writing different types of tests.

In addition to the preceding chapters, the following appendices of this guide provide additional information useful to test developers when creating custom tests and test suites for Java ME technology implementations:

- Appendix A describes the test APIs for different types of tests.
- Appendix B describes the contents of the Framework redistributables directory.
- Appendix C provides a summary of Framework keywords and test description fields.

Test developers can also refer to the *JavaTest Architect's Guide* for more in-depth explanations about creating test suites that run on the JavaTest harness. The *JavaTest Architect's Guide* is divided into two parts. Part I, *The Basics*, is useful for aspiring test suite architects and includes basic topics such as a tutorial that introduces the JavaTest GUI as well as descriptions of test suite components and basic processes of creating test suites, tests, and configuration interviews that work with the JavaTest harness. Part II, Advanced Topics, includes more in-depth information about working with the JavaTest harness such as developing test scripts that run tests and using JavaTest harness standard commands that configure an environment for running test suites on specific test platforms.

Using the Framework

The Framework is a bundled set of resources used by test suite developers and test writers to create tests and test suites for Java ME technology implementations as well as to provide a bridge between the device and the harness when users run tests.

Java ME technology tests and test suites are run on a device by the harness. Because of Java ME technology connectivity requirements (see "Connectivity Requirements" on page 4) and limited hardware resources test devices (see "Resource Limitations" on page 5), test suite and test developers are presented with a number of challenges. To help simplify test development, the Framework provides a set of components (harness plug-ins and support classes) for developers. When running tests, the Framework and its components act as a bridge between the harness (see "Framework Components on the Harness Side" on page 6) and a test device (see "Framework Components on the Device Side" on page 7).

The Framework resources for both the harness host and the test device enable developers to reduce the complexity of test suite development while optimizing test performance. Because Framework classes and resources are shared by multiple test suites, they are fully developed, extensively tested, and stable. In addition to reducing the complexity of Java ME test suite development, Framework classes and resources can improve the reliability of the test suite.

Development Environment

The following tools and resources are the minimum software requirements for the Framework development environment:

- JDK, version 5.0 or later
- JavaTest harness, version 3.2.2 or later

For TCK development, download the latest Java Compatibility Test Tools (Java CTT) from the Java Community Process (JCP) program web site.

Target Environment

The Framework resources enable developers to package and deliver tests developed for a device in an appropriate form for a particular platform implemented on a device. The Java ME application environment includes both a configuration such as Connected Limited Device Configuration (CLDC) or Connected Device

Configuration (CDC) and a profile such as Mobile Information Device Profile (MIDP), Foundation Profile (FP), Personal Basis Profile (PBP), or Personal Profile (PP).

- **Configuration -** CLDC and CDC are configurations that provides a basic set of libraries and virtual-machine features that must be present in an implementation of a Java ME environment.

  When coupled with one or more profiles, the configuration provides developers with a stable Java platform for creating applications for consumer and embedded devices. Each configuration supports optional packages that enable product designers to balance the functionality needs of a design against its resource constraints.
- **Profile** - A set of standard APIs that support a category of devices for a specific configuration.

  A specific profile is combined with a corresponding configuration to provide a complete Java application environment for the target device class.
- **Optional packages** - A set of technology-specific APIs that extends the functionality of a Java application environment.

  Optional packages provide specific areas of functionality.

The ability to specify bundles enables test developers to match software and hardware capabilities. They can use APIs that provide easy access to the components that a device has, without the overhead of APIs designed for capabilities the device doesn't support.

Connectivity Requirements

Each Java technology has a unique set of connectivity requirements. When using the Framework resources to develop test suites, developers should consider the following connectivity requirements:

- **CLDC (without MIDP)** - No connectivity required by specification; however, the Framework requires bi-directional communication.
- **MIDP** - HTTP is required.
- **CDC** - Datagram connection is optional in the specification.The Framework requires bi-directional communication (Datagram, TCP/IP, HTTP, serial, or custom communications are supported).
- **FP on CDC** - Full TCP/IP is optional in the specification. The Framework requires bi-directional communication (Datagram, TCP/IP, HTTP, serial, or custom communications are supported).

Resource Limitations

Hardware resources on test devices are often limited. Resource constrained devices can quit operation when excess native resources are requested. The Framework can run tests on resource constrained target devices with limited available memory and persistent storage. When developing test suites for the Java ME technology device, the developer must be aware of device limitations such as the following and use the appropriate Framework resources:

- Memory constraints of the device
- Minimum requirements listed in the appropriate specification for profiles
- Maximum number of connections (HTTP, SSL, or Socket TCP/IP) on the networking subsystem that can be open at any one time
- Graphics and image subsystem limits

CLDC Target Device

The CLDC configuration provides a Java platform for network-connected devices that have limited processing power, memory, and graphical capability (such as, cellular phones, pagers, low-end personal organizers, and machine-to-machine equipment). CLDC can also be deployed in home appliances, TV set-top boxes, and point-of-sale terminals.

CLDC target devices typically have the following capabilities:

- A 16-bit or 32-bit processor with a minimum clock speed of 16 megahertz
- At least 160 kilobytes of non-volatile memory allocated for the CLDC libraries and virtual machine
- At least 192 kilobytes of total memory available for the Java platform
- Low power consumption, often operating on battery power
- Connectivity to a network, often through an intermittent wireless connection with limited bandwidth

CDC Target Device

The CDC configuration provides a Java platform for network-connected consumer and embedded devices, including smart communicators, pagers, high-end PDAs, and set-top boxes.

CDC target devices typically have the following capabilities:

- A 32-bit microprocessor or controller
- 2 megabytes of RAM available to the Java application environment
- 2.5 megabytes of ROM available to the Java application environment

Framework Components

The Framework provides a set of existing components that developers can include in a test suite to create a bridge between a workstation or PC running the harness and a device containing the application under test. As a bridge, Framework components plug into the harness and the device.

Framework Components on the Harness Side

The following components are used for running CLDC, MIDP, and CDC tests in the Distributed, OTA, and Automated test configurations with the harness:

- **Execution server -** Used in CLDC and MIDP Distributed and Automated test configurations.

  The execution server contains no internal test-related logic. Its only function is to forward data. It is as lightweight as possible.
- **Test provider -** In addition to the execution server, a test provider acts as a server to the execution server.

  The execution server knows its test provider and calls its methods to pass the data from the client to the test provider and vice versa.
- **OTA provisioning server -** Used in the MIDP OTA test configuration.

  The OTA provisioning server supplies applications over the air to wireless devices.
- **Passive agent -** Used in CLDC, MIDP, and CDC Distributed and OTA test configurations.

  An agent is a Java SE side component that works in conjunction with the harness to run server-side parts of the tests on Java SE, on the same or different system that is running the harness. Passive agents wait for a request from the harness before running tests.
- **Server-side test -** Used in CLDC, MIDP, and CDC Distributed and OTA test configurations.
- **Messaging service -** Used in CLDC, MIDP, and CDC Distributed test configuration.
- **Interview classes and support classes -** Used in CLDC, MIDP, and CDC to create interviews.

Framework Components on the Device Side

The device used to run tests might be a resource constrained device in which available memory and persistent storage are limited. The Framework includes the following components for running CLDC, MIDP, and CDC tests on a device in the Automated, Distributed, and OTA test configurations:

- **AMS** - Used in CLDC and MIDP. Application management code required on the target device to receive the bundle with the test execution agent and the tests from the harness is called the Application Management Software (AMS).

  In some contexts, AMS is referred to as Java Application Manager (JAM).

- **Agent** - Used in CLDC, MIDP, and CDC Automated and Distributed test configurations.

  An agent is a separate program that works in conjunction with the harness to run tests on the device. Agents can be either active or passive.

  - **Active agents** - Are used when the agent must initiate the connection to the JavaTest harness.

    Agents that use active communication enable users to run tests in parallel by using multiple agents simultaneously and to specify the test machines at the time the tests run. If the security restrictions of a test system prevent incoming connections, you must use an active agent.

  - **Passive agents** - Are used when the agent must wait for the JavaTest harness to initiate the connection.

    Because the JavaTest harness only initiates a connection to a passive agent when it runs tests, passive communication requires that the test machine is specified as part of the test configuration (not at the time the tests run) and does not enable running tests in parallel. Passive agents must be started before the harness attempts to run tests.

  In CDC, the agent is started once, and when started, in a loop, it uses the communication channel to the harness to download the test classes and to execute them on the device, and then reports back the results. In CLDC and MIDP configurations, the execution server supplies the test bundle (it includes the test agent and the tests) and the device's AMS fetches the bundle and then executes the test agent which in turn executes the bundled tests and reports the results back to the harness.

- **Tests** - Used in CLDC, MIDP, and CDC Automated, Distributed, and OTA test configurations.

  The source code and any accompanying information that exercise a particular feature, or part of a feature, of a technology implementation to make sure that the feature complies with the Java specification. A single test can contain multiple test cases. Accompanying information can include test documentation, auxiliary data files, or other resources required by the source code. Tests correspond to assertions of the specification.

Test Types

The developer uses the Framework resources for and organizes the tests based on the test type or testing function (for example, automated tests must be grouped separately from interactive tests because they use different Framework resources). TABLE 1-1 presents a simple matrix of the different test configurations and the types of tests that the Framework resources support.

TABLE 1-1 Configurations and Supported Test Types

Configuration	Automated Tests	Distributed Tests	Interactive Tests	OTA Tests
CLDC (without MIDP)	Supported	Unsupported	Unsupported	Unsupported
MIDP	Supported	Supported	Supported	Supported
CDC	Supported	Supported	Supported	Unsupported

Automated Tests

Automated tests for CLDC, MIDP, and CDC configurations execute on the test device without requiring user interaction. Automated tests can be queued up and run by the test harness and their results recorded without the user being present.

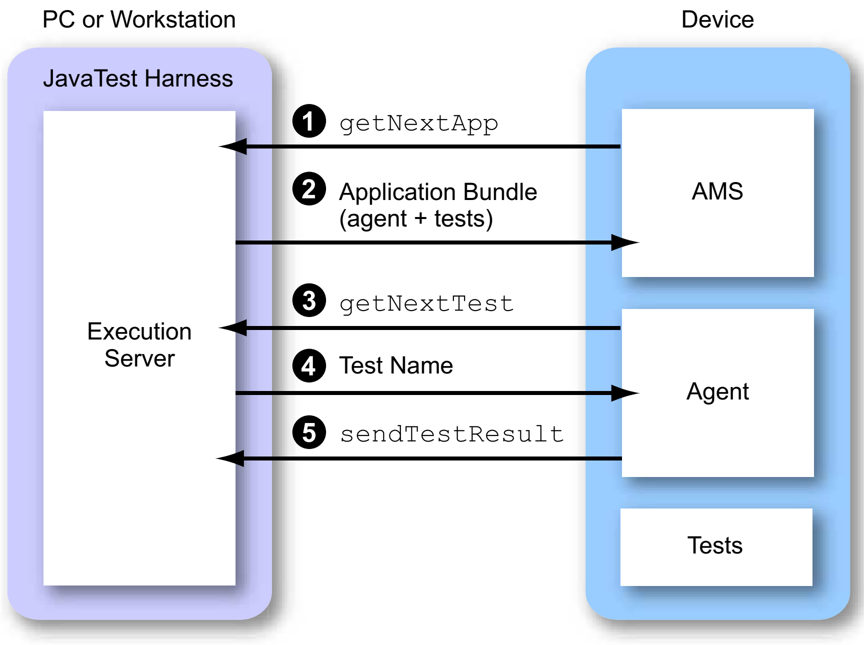
The configuration for standard automated test execution is the most common and the most simple of the tests that are run on the device. Automated tests are also the most convenient and the fastest tests for a user to run, since they are fully automated. The majority of tests in a test suite should be automated tests with other types of tests used only when it's impossible to write automated tests.

In CLDC and MIDP configurations, the harness (running on a PC or a workstation) sends an application bundle containing the tests and an agent to the device where they are unpacked by the application management software (AMS) built into the device and run. In this configuration, user interaction is not required to run each test.

FIGURE 1-1 illustrates the Framework configuration for running CLDC and MIDP standard automated tests. For CDC, the agent is started, downloads the tests via the communication channel, and executes them, without being restarted (a single agent runs from the beginning to the end of the test run).

See Chapter 3 for an example of writing an automated test and "Testing Devices With Automated Tests" on page 47 in Chapter 5 for information about automated test execution.

FIGURE 1-1 Framework Configuration for Standard Automated Tests



Distributed Tests

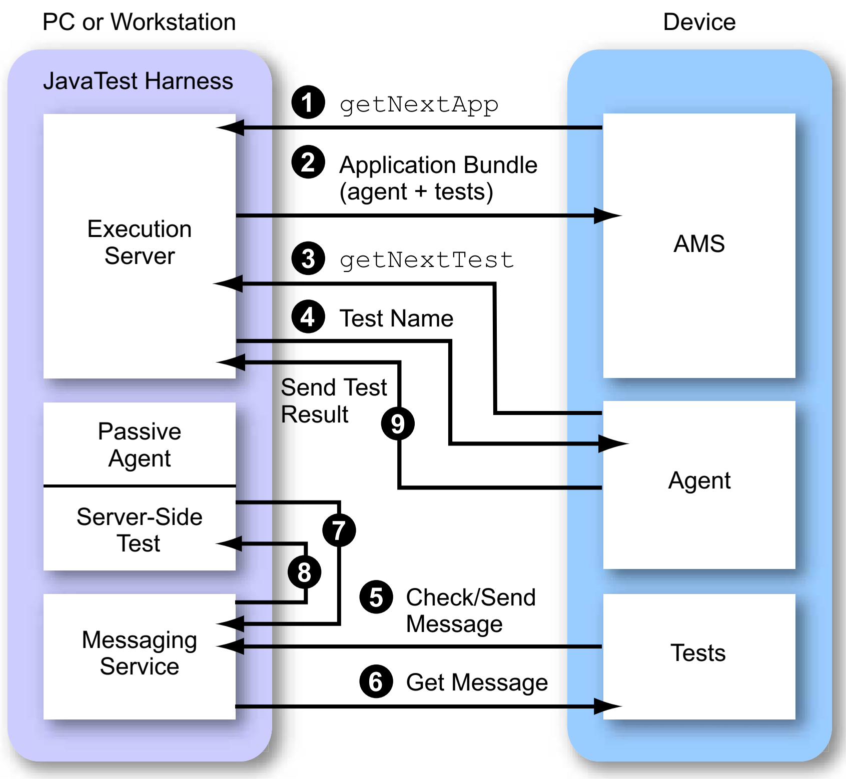
Distributed tests are a special type of automated tests. Not only do they have a device side test component, which is executed under the control of a test agent (as with any regular automated tests), but they also have one or more remote components on other devices or the Java SE platform side. The distributed test components have names and communicate with each other by sending messages to each other by way of a messaging service. The remote components of a distributed test typically run on a harness host by using a passive agent in the same virtual machine as the harness and provide some additional functionality needed by the test. For example, a test verifies that an HTTPS connection can be made to the remote host. The remote component that runs on the Java SE platform would be an HTTPS server. The test on the device performs the following sequence of actions:

1. Sends a message requesting that the server start.
2. Connects to the server and verify that the connection is OK
3. Sends a message to stop the server

Distributed tests are typically slower (due to extra communication between remote components) and more complex than simple automated tests and should be used only when it's not possible to write simple automated tests.

FIGURE 1-2 illustrates the Framework configuration for running CLDC and MIDP distributed tests. Distributed tests are currently not supported in CLDC (without MIDP). For CDC, the agent is started, downloads the tests via the communication channel, and executes them, without being restarted (a single agent runs from the beginning to the end of the test run). See "Testing Communications or Networking With Distributed Tests" on page 49 in Chapter 5 for information about writing distributed tests and distributed test execution.

FIGURE 1-2 Framework Configuration for Distributed Tests



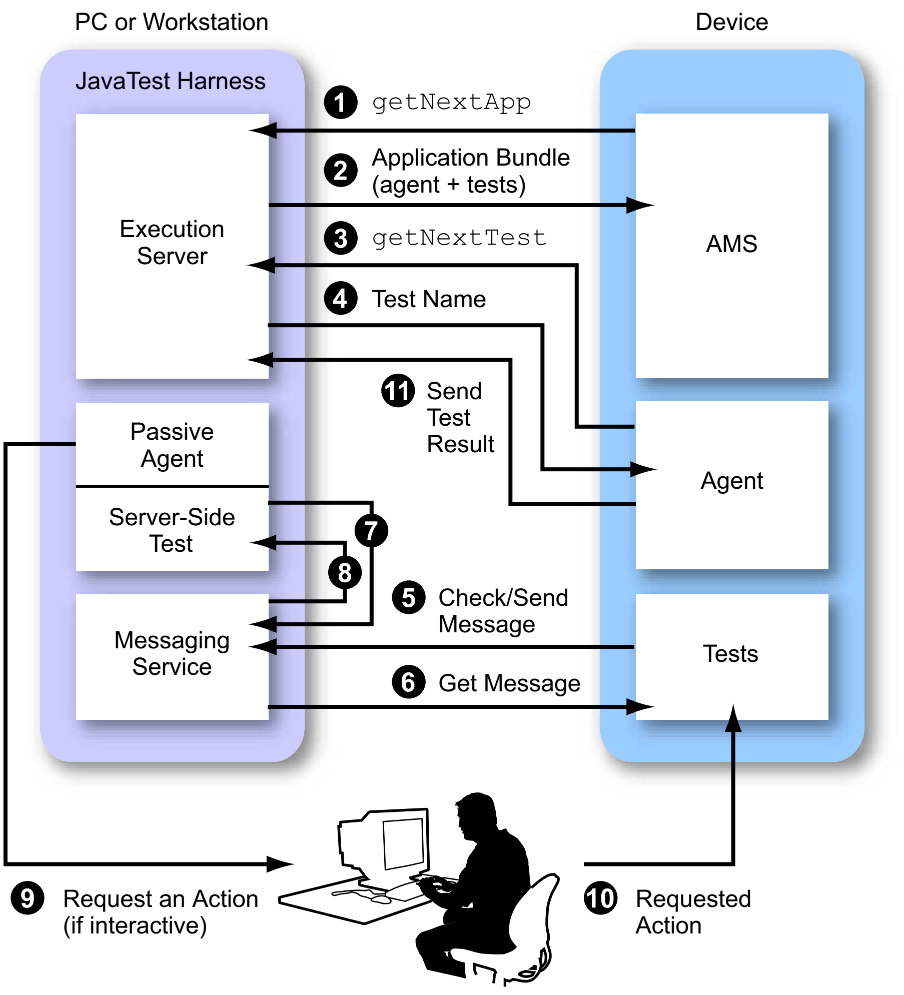
Interactive Tests

Interactive tests are the tests that require some form of user interaction and cannot be executed without such interaction. From a design point of view, interactive tests are a subtype of distributed test. As a subtype of distribute test, interactive tests generally execute on the test device under the control of a component called an agent. However, unlike distributed tests, interactive tests also require some form of user interaction as a part of the test. Interactive tests might require that the user change something with a device, which triggers event generation or require the user to verify that a device plays sound or vibrates. But most typically, interactive tests are used to validate user interface on the device.

Like distributed tests, interactive tests validate API functionality while the device is connected to a remote host (the PC or workstation where the harness runs). In this configuration, one part of the distributed test runs on the device and the other part of the test runs on a remote host (the PC or workstation where the harness runs) using a passive agent running on in the same VM as the harness.

Interactive tests are not supported in pure CLDC (without MIDP). FIGURE 1-3 illustrates the Framework configuration for MIDP interactive tests. For CDC, the agent is started, downloads the tests via the communication channel, and executes them, without being restarted (a single agent runs from the beginning to the end of the test run).See "Testing User Interfaces With Interactive Tests" on page 54 in Chapter 5 for information about writing interactive tests and interactive test execution.

FIGURE 1-3 Framework Configuration for Interactive Tests



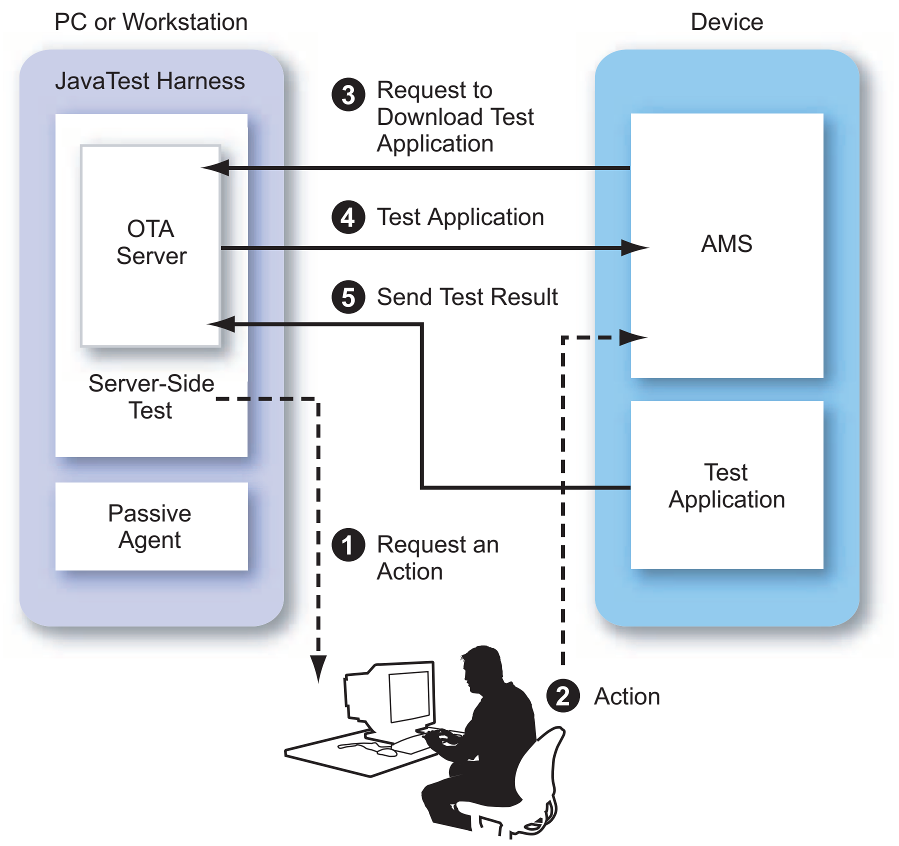
OTA Tests

OTA tests are MIDP-specific tests that verify the over-the-air (OTA) application provisioning implementation. This includes obtaining, installing, and removing applications (MIDlet suites), and enforcing security requirements. Each OTA test has an associated MIDlet suite that you download from the provisioning server and install and launch on the test device. Multiple instances of each can run in parallel, sharing one OTA server.

OTA tests validate API functionality while the device is connected to a remote host (the PC or workstation where the harness runs). In this configuration, one part of the OTA test runs on the remote host (the PC or workstation where the harness runs) using a passive agent and the other part of the test runs on the device. OTA tests require user interaction as a part of each test.

FIGURE 1-4 illustrates the Framework configuration for running OTA tests. See "Testing Application Provisioning With OTA Tests" on page 59 in Chapter 5 for information about writing OTA tests and OTA test execution.

FIGURE 1-4 Framework Configuration for OTA Tests



CHAPTER 2

Installation

This release of the Framework contains binaries and source code bundled in a `.zip` file. This chapter describes the procedure required to install the Framework bundle on a development system.

This chapter contains the following sections:

- Prerequisites to Installing the Framework Bundle
- Installing the Framework Bundle
- Installed Directories and Files

Prerequisites to Installing the Framework Bundle

The following tools and resources are the minimum software that should be installed before installing the Framework bundle on a development system:

- **Java Development Kit** - The commercial version of Java Development Kit (JDK™) version 5 (also known as JDK version 1.5) or later is required.

  Download the JDK software from `http://java.sun.com/javase/downloads` and install it according to the instructions on the web site.

- **JavaTest harness** - Version 3.2.2 or later is required.

For TCK development, download the latest Java Compatibility Test Tools (Java CTT) from the Java Community Process (JCP) program web site.

Installing the Framework Bundle

The Java ME TCK Framework resources are packaged and provided to users as a zip bundle. Two main tasks are performed when installing the development kit:

- Downloading the `.zip` file.
- Unzipping the Framework bundle.

▼ To Install the Framework Bundle

Before installing the Framework bundle, verify that JDK version 5 or later is installed on the development system. See "Prerequisites to Installing the Framework Bundle" on page 15 **for the download location and installation instructions of the JDK.**

1. **Download the Framework `.zip` file to a directory on your system.**
2. **Unzip the bundle into an empty directory of your choice.**

   The directory that contains the unzipped Framework directories and files becomes the Java ME TCK Framework root directory.

Installed Directories and Files

The Framework bundle installs the following files and directories containing the binary files and source code:

- `doc` - Contains PDF and HTML versions of this *Java ME TCK Framework Developer's Guide*.
- `redistributables/` - Contains the following directories:
  - `javame_tck_framework_121/src` - Contains the source files.
  - `javame_tck_framework_121/lib` - Contains compiled Java ME TCK Framework classes prepackaged into Java Archive (JAR) files.
  - `javame_tck_framework_121/doc/javame_tck_framework/api/`- Contains the Framework API documentation.

  See Appendix B for a detailed description of the contents of the `redistributables` directories.
- `samples` - Contains the following directories:

- samples/binaries - Contains prebuilt SimplestTestSuite, SimpleTestSuite, and AdvancedTestSuite sample test suites ready for use.
- samples/sources - Contains sources and build files required to build the SimplestTestSuite, SimpleTestSuite, and AdvancedTestSuite sample test suites either as an integrated part of the main ME Framework build or as a stand-alone project.

  The SimplestTestSuite directory provides sources for the minimum set of classes required to create a test suite that executes automated tests.

  The SimpleTestSuite directory provides sources required to create a test suite that executes automated and distrubuted tests in MIDP and CDC mode. The examples in this guide use files contained in this directory.

  The AdvancedTestSuite directory provides sources required to create a typical conformance test suite.
- See "Building an Updated Simple Test Suite" on page 23 for a description of the procedure used to build a standalone sample test suite. To build the sample test suite as part of the ME Framework, use the ant samples command in the procedure.

Note – Test developers use the sample code contained in samples/sources/SimpleTestSuite when following the examples in this guide.

- ReleaseNotes-me_framework.html - Contains additional information about the Java ME TCK Framework release.
- COPYRIGHT-me_framework.html - Contains the copyright notice for the Framework.
- index.html - Contains descriptions of the Java ME TCK Framework bundles as well as hyperlinks to user documentation provided by the Framework.
- document.css - Style sheet used by HTML files.

CHAPTER 3

Writing a Simple Automated Test

Automated test execution is the most common and simple of the test configurations that are run on the test device. User interaction is not required to run automated tests. The following instructions describe how to create a simple automated test and add it to an existing Framework supplied test suite.

This chapter contains the following sections presented in the sequence that a test developer follows when writing a test that is added to an existing test suite:

- Writing an Automated Test
- Building an Updated Simple Test Suite
- Testing an Updated Simple Test Suite

Writing an Automated Test

This chapter provides the source code and instructions for creating a simple automated test. General information about the process of creating tests can be located in Chapters 5 and 6 of the *Java Technology Test Suite Development Guide*.

When creating an automated test, test developers must consider the following factors:

- Automated tests must extend `com.sun.tck.cldc.lib.MultiTest`, a base class for tests with multiple sub test cases.

  This example is valid for CLDC, MIDP, and CDC automated tests. For tests that are to be executed only on a CDC stack and never executed on CLDC/MIDP, the base class is `javasoft.sqe.javatest.lib.MultiTest`.

- Developers must add individual test case methods to the derived test class to create a useful test class.

- Each test case method must take no arguments, and must return a Status object that represents the outcome of the test case. The Status can be either `Status.passed(String)` or `Status.failed(String)`.
- Developers must update the `runTestCases()` method to add the test cases that are executed.

  This is a CLDC-MIDP specific method (the abstract method is defined in CLDC-specific `MultiTest`). For CDC-specific Multitest, this method is not required because reflection is available on CDC stacks.
- Each test in a test suite has a corresponding test description that is typically contained in an HTML file. The test description file contains all information required to run the test, such as the source file to use (`source`), the class to execute (`executeClass`), and how the test is executed (`keyword`).

▼ To Create a Simple Automated Test

The following steps demonstrate how to create an automated test class suitable for CLDC, MIDP, and CDC (`Test3.java`), to create its test description file (`index.html`), and to update the test class dependencies file (`testClasses.lst`).

The test class, `Test3.java`, is a simple automated test class. This sample uses files from the `SimpleTestSuite` located at `samples/sources/SimpleTestSuite/` (not the `SimpleTestSuite` or the `AdvancedTestSuite`).

This test class does not test a particular API. Instead, it is used to demonstrate the following aspects of creating a test class:

- The format of an automated test class
- How a test case method is implemented
- How a test case is selected in the `runTestCases()` method
- How each test case returns a `Status` object
- How to use `ref` to output reference information for debugging purpose

A text editor or an integrated development environment (IDE) of your choice is the only tool required to develop a test.

1. **Enter the following test code into a text editor or IDE of your choice.**

```
package sample.pkg3;

import com.sun.tck.cldc.lib.MultiTest;
import com.sun.tck.cldc.lib.Status;

/**
* Sample test with simple test cases.
```

```
*/
public class Test3 extends MultiTest {

   protected void runTestCases() {
      if (isSelected("helloWorld")) {
         addStatus(helloWorld());
      }
 }

 public Status helloWorld() {
     String message1 = new String("Hello World !");
     String message2 = new String("Hello World !");
     ref.println("message1: "+message1);
     ref.println("message2: "+message2);
     if (!message2.equals(message1)) {
         return Status.failed("Failed: see ref for details");
     }
     return Status.passed("OK");
   }
}
```

2. **Save this file in the Simple Test Suite source as** `SimpleTestSuite/tests/pkg3/Test3.java`**.**

   The `pkg3` folder does not exist in `SimpleTestSuite/tests` and can be created by the test developer when saving `Test3.java`. Sources for the tests should be placed inside the `tests` directory of the test suite and organized by package.

3. **Enter the following HTML code into a text editor or IDE of your choice.**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Test Specifications and Descriptions for Test3</TITLE>
</HEAD>
<BODY>
<H1>Test Specifications and Descriptions for Test3</H1>
<HR>
<a name="Test3"></a>
<TABLE BORDER=1 SUMMARY="JavaTest Test Description" CLASS=TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH SCOPE="col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> Hello World ! test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test3.java">Test3.java</A> </TD>
</TR>
```

```
<TR>
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg3.Test3 </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD>runtime positive </TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

Note – The contents of this test description file correspond to the `Test3.java` test class.

---

4. **Save this file in the Simple Test Suite source as `SimpleTestSuite/tests/pkg3/index.html`.**

   The `index.html` must be contained in the same directory as its test class (in this case, `Test3.java`).

5. **Update the test class dependency file (`testClasses.lst`) in the build directory (`SimpleTestSuite/build`) by adding the following line that identifies the new test and its class.**

```
sample/pkg3/index.html#Test3 sample/pkg3/Test3.class
```

   The test class dependency file (`testClasses.lst`) provides information to the Framework test bundling infrastructure regarding which classes should be bundled for each test. This information is used in CLDC and MIDP, but ignored in CDC.

Note – In Framework version 1.2, the test bundling mechanism was improved so that in simple cases such as this (the complete test is in a single file without using other test classes) updating the `testClasses.lst` file is not required.

Additional Action

After creating the new test class, creating the test description file, and updating the test class dependency file, you must build (see "Building an Updated Simple Test Suite" on page 23) and test the updated test suite (see "Testing an Updated Simple Test Suite" on page 24) before making the test suite available to users.

Building an Updated Simple Test Suite

After completing the procedures contained in “Writing an Automated Test” on page 19, you must run the build to assemble the updated test suite.

▼ To Build an Updated Test Suite

The following steps demonstrate how to build an updated test suite after adding a test.

Before building the updated test suite, your development environment must meet the minimum requirements described in “Development Environment” on page 3. You must also have the following tools installed to build the updated Simple Test Suite:

- Ant 1.6.5
- Java ME TCK Framework, version 1.2.1
- JavaTest harness, version 3.2.2
- WTK (Wireless Toolkit), version 2.5 or 2.5.1

1. **Uncomment and modify the four required properties in** `SimpleTestSuite/build/build.properties` **file, in accordance with your environment.**

   The following is an example of changes for a Windows environment.

```
ME_FRAMEWORK_LIB_DIR=D:\\javame_tck_framework_121\\lib
WIRELESS_TOOLKIT=D:\\WTK25
JTHARNESS_JAR=D:\\javatest-3.2.2\\lib\\javatest.jar
JAVAHELP_JAR=D:\\javatest-3.2.2\\lib\\javatest.jar
```

   If using Linux to build a test suite, refer to the system requirements section of the appropriate WTK documentation for required configuration information.

Note – When the commercial version of the JavaTest harness is used, `JAVAHELP_JAR` should point to the commercial version of the `javatest.jar` file. There is no need to download the JavaHelp binaries, since they are present inside the `javatest.jar` file.

---

2. **Use a terminal window or console to make the** `build` **directory your current directory.**

3. **Enter the `ant` command to invoke the ant build script.**

   The build creates a `SimpleTestSuite-build` directory containing the test suite.

Additional Action

After creating the updated test suite, you must test it (see "Testing an Updated Simple Test Suite" on page 24) before making the test suite available to users.

Testing an Updated Simple Test Suite

After completing the procedures contained in "Building an Updated Simple Test Suite" on page 23, you must test the updated test suite by executing it with the JavaTest harness.

▼ To Test an Updated Test Suite

The following steps demonstrate how to test an updated test suite. General information about testing can be located in Chapter 7 of the *Java Technology Test Suite Development Guide*.

Before executing the test suite, your development environment must meet the minimum requirements described in "Development Environment" on page 3. You must also have the following tools installed to test the updated Simple Test Suite:

- Java ME TCK Framework 1.2.1
- JavaTest harness 3.2.2
- Wireless Toolkit (WTK) version 2.5

Note – The following commands are shown using Microsoft Windows system prompts and syntax.

---

1. **Make the Simple Test Suite root directory your current directory.**

2. **Start the JavaTest harness by using the following command.**

```
java -jar lib\javatest.jar -newDesktop
```

   The JavaTest harness displays either the Welcome to JavaTest dialog box or the Quick Start wizard. See the JavaTest harness documentation for detailed information about using the JavaTest harness.

3. **Use the Configuration Editor to provide configuration information required to run the updated test suite.**
4. **After you complete the interview, choose Run Test > Start on the Test Manager main menu to start the test suite.**
5. **Use the following command to start the WTK.**

```
c:\WTK_InstallDir\bin\emulator ^
    -Xautotest:http://%JAVATEST_HOST%:%JAVATEST_PORT%/test/getNextApp.jad
```

If you answered the interview questions correctly, all of the tests run successfully.

CHAPTER 4

Constructing a Test Suite

This chapter describes the organization and construction of a Java ME technology test suite that uses Framework resources. Additional information about constructing test suites for use with the JavaTest harness can be located in Chapters 4 and 8 of the *JavaTest Architect's Guide*.

This chapter contains the following sections:

- Test Suite Structure
- Creating and Using a Configuration Interview
- Building a Test Suite

Test Suite Structure

Test suites are the main unit of test development and deployment. A test suite is a self-contained collection of tests designed to test a major feature or a major subset of an API or a profile. When architects and developers define the contents and structure of a test suite, they should group tests that use the same test setup to interact with the test device. Grouping tests in this way enables users to run all tests in the test suite without changing the test setup.

For example, tests for a profile might be divided into two types. One type of test exercises the technology's API implemented on the test device. The other type of test exercises the technology implementation's interaction with an Over the Air (OTA) server. Because the test setup for these two kinds of tests is substantially different, the architect and developer might group these tests into two independently run test subsets to make them easier for the user to configure and run.

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

- `testsuite.jtt` file

- `lib` directory
- `tests` directory
- `classes` directory
- `doc` directory

`testsuite.jtt` File

The `testsuite.jtt` file is located in the root directory of the test suite. It provides a registry of information about the test suite and defines test suite properties. The harness uses these properties to instantiate a `TestSuite` object that acts as a portal to all information about the test suite. Whenever the harness requires information about the test suite, it queries the `TestSuite` object.

The `testsuite.jtt` file generally contains the following entries that tell the JavaTest harness how to start the `TestSuite` class. It might also contain other entries. See the *JavaTest Architect's Guide* for detailed information about the standard properties used by the `TestSuite` that can be specified in the `testsuite.jtt` file.

- `name` - The name of the test suite.
- `id` - A unique identifier composed of letters, digits, underscore, minus, and hyphen used to identify a specific version of a test suite.
- `tests` (optional) - By default, the JavaTest harness looks for test source files and test descriptions in the `tests/` directory in the test suite root directory.
- `classpath` - Entry that specifies the class path on which the main `TestSuite` class can be found (typically, a JAR file that contains test suite-specific components).

  This entry is required if the `TestSuite` class or any other classes the `TestSuite` refers to are not located within `javatest.jar`.
- `testsuite` - Entry that specifies the name of the test suite class and any arguments that the class requires.

The following is the `testsuite.jtt` file used by the Simple Test Suite that comes with the Framework.

CODE EXAMPLE 4-1 Simple Test Suite `testsuite.jtt` File

```
name=Simple Test Suite
id=Sample_TestSuite_1
tests=tests
classpath=lib/j2mefw_jt.jar lib/sample_jt.jar lib/interviewlib.jar
testsuite=sample.suite.SampleTestSuite
```

The `testsuite.jtt` file is located under the root directory of the Simple Test Suite. You can also find it under the `build/` directory of the Simple Test Suite source.

`lib` Directory

The `lib` directory usually contains the `javatest.jar` file that provides all of the classes required to execute the harness, all of the JAR files under the `lib` directory of the Java ME TCK Framework, and the library classes. The test suite developer can use the library classes to simplify the creation of tests. With `javatest.jar` in the `lib` directory, the harness automatically locates the test suite and does not prompt the user for the path to test suite directory.

This directory also contains additional resource files required by the test suite. These files might include the following:

- `testsuite.jar` - If a custom interview or if customized harness plug-in classes are used, package the classes and interview files in a custom `testsuite.jar` file and place it in the `lib` directory.

  In the Simple Test Suite example, this file is named `sample_jt.jar` and located under the `SimpleTestSuite/lib` directory.

- `testsuite.jtx` - The exclude list (`testsuite.jtx`) file identifies the tests in a test suite that are not required to be run.

  Tests are not excluded in the Simple Test Suite, so it does not contain a `testsuite.jtx` file.

  The exclude list file usually has the following format:

  Test-URL[*Test-Cases*] *BugID Keyword*

  The following is an example of two lines from an exclude list file.

```
api/java_awt/EventQueue/index.html#Invoke[EventQueue2006] 6406330 test
api/java_awt/Adjustable/index.html#SetGetValue 4682598 spec_jdk
```

`tests` Directory

The `tests` directory contains test sources and test descriptions grouped by the test developer according to a specific design principle. Tests might be grouped in a test directory in the following cases:

- All tests that examine the same component or functionality
- All tests that have a configuration in common

Organize the tests hierarchically the way you want them displayed in the test tree. The Test Manager in the harness displays the test hierarchy, enabling users to select and run specific groups of tests from the GUI.

Test Class

A test class or test source is a Java technology class that either implements the test interface or extends the test class. A test class can rely on inner, sub, or independent classes and contains one or more test cases. Users must add individual test case methods to the derived test class to create a useful test class.

See Chapter 5 for information required to write different types of tests for the Framework.

Test Case

A test case represents a single test and is the smallest test entity. If a test class defines only one test case, the test class is equivalent to the test case. Each test case must return a Status object that represents the outcome of the test case.

The following example shows a very simple test class which contains several test cases.

CODE EXAMPLE 4-2 Simple Test Class

```
public class MyTest extends MultiTest {

   protected void runTestCases() {
      if (isSelected("testCase1")) {
         addStatus(testCase1());
      }
      if (isSelected("testCase2")) {
         addStatus(testCase2());
      }
   }

   public Status testCase1() {
       if (1 + 1 == 2)
           return Status.passed("OK");
       else
           return Status.failed("1 + 1 did not make 2");
       }
   }

   public Status testCase2() {
       if (2 + 2 == 4)
```

CODE EXAMPLE 4-2 Simple Test Class *(Continued)*

```
            return Status.passed("OK");
        else
            return Status.failed("2 + 2 did not make 4");
        }
    }
}
```

Additional examples (`Test1.java` and `Test2.java`) can be found in the following Simple Test Suite directories:

- `SimpleTestSuite/tests/sample/pkg1/`
- `SimpleTestSuite/tests/sample/pkg2/`

Test Description File

Each subdirectory that contains one or more test classes must also contain a corresponding test description. The test description is contained in HTML format.

The test description file generally contains the following fields:

- `title` - A descriptive string that identifies what the test does.

  The title appears in reports and in the harness status window.
- `source` - List of source files of the test.

  When the test sources are listed in this field, they can be accessed and viewed from the harness GUI.
- `executeClass`- Specifies the name of the test's executable class file.

  It is assumed to be located in the classes directory.
- `keywords` - String tokens that can be associated with a given test.

  They describe attributes or characteristics of the test (for example, how to execute the test, and whether it is a positive or negative test). Keywords are often used to select or deselect tests from a test run. The most common keywords are `runtime` and `positive`. See Chapter 5 for a description of the specific keywords required for the different types of tests that can compose a test suite.

See Appendix C for a summary list of the Framework test description fields and keywords.

`classes` Directory

The `classes` directory contains all of the compiled test classes and library classes required to run the tests.

The classes directory generally contains the following sub-directories:

- classes/preverified - The preverified test classes.
- classes/shared/testClasses.lst - The test class dependency file (testClasses.lst) that provides information to the Framework test bundling infrastructure regarding which classes should be bundled for each test.

  This information is used in CLDC and MIDP, but ignored in CDC.

doc Directory

The doc directory contains test suite-specific documentation such as User Guides that describe how to run the test suite.

Creating and Using a Configuration Interview

All nontrivial test suites require additional information about the tests in order for the test harness to execute them. This additional information, referred to as the test configuration, is obtained from the user through a test suite specific-configuration interview written by the test developer. Additional information about creating configuration interviews for use with the JavaTest harness can be located in Chapter 6 of the *JavaTest Architect's Guide*.

The configuration interview consists of a series of questions about the test configuration or specific target implementation, which the user must answer before running tests. The Configuration Editor displays the configuration interview in the harness and exports the user's answers in a test environment object from which the harness and the Framework can access and obtain the data.

For example, if a test must get the hostName and the portNumber during the test run, the following configuration conditions must be satisfied for the test to run:

- The test description file (see "Test Description File" on page 31) must include an appropriate executeArgs argument.

  The following is an example of how executeArgs might be specified in the test description file.

```
<tr>
<td scope="row"> <b>executeArgs</b> </td>
<td> -host $testHost -port $testPort </td>
</tr>
```

In the test description file, `$testHost` and `$testPort` are test environment variables and are replaced by actual values obtained from the test environment.

- The test suite configuration interview must include a corresponding and appropriate question asking the user to specify the values of the host and port number.

  The configuration interview creates entries in the test environment from user answers as name-value pairs. The value of `$testHost` and `$testPort` are defined in the configuration from those user answers. Users can display the test environment from within the harness by choosing Configure > Show Test Environment from the Test Manager menu bar.

  The following is an example of name-value pairs that the configuration interview might create in a configuration file from user answers.

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

Creating a Configuration Interview

There are two ways of creating a configuration interview for an ME Framework based test suite:

- The simpler but less customizeable approach is to use the `com.sun.tck.j2me.interview.BasicTckInterview` class.

  The repository contains an example test suite (`SimpleTestSuite`) that uses this approach. In this example, the TestSuite object creates an interview by means of the Builder Pattern. The sample also demonstrates how to add additional questions to a custom sub-interview. See "To Create a Configuration Interview Through the Interview Class" on page 34.

- A more complex but more flexible approach (used by many complex Java ME TCKs) creates a custom test suite interview by extending the `com.sun.tck.j2me.interview.MidpTckBaseInterview` class.

  To extend `com.sun.tck.j2me.interview.MidpTckBaseInterview`, you must write questions (such as `name` and `description)` as well as override some base methods. The `samples` directory contains an example test suite (`AdvancedTestSuite`) that uses this approach.

Note – To get up and running quickly, start with the first approach and switch to the second approach when your test suite requires a more advanced configuration. In most cases, the first approach is sufficient to configure and run a test suite.

After creating an interview, you must plug your `testsuite.jtt` file, TestSuite class, and Interview class into the JavaTest harness to run the tests. See "To Plug In a Custom Interview" on page 44.

▼ To Create a Configuration Interview Through the Interview Class

The following procedure uses the `Test2.java` test class as an example to demonstrate how to accomplish the following tasks:

- Create a configuration interview through the `interview` class
- Export an environment variable
- Specify the environment value by using `context` or `executeArgs`
- Decode the argument into the test class

`SimpleTestSuite/tests/sample/pkg2/Test2.java` is a simple test class created to demonstrate the basic principle of writing a test that uses a configuration interview.

In addition to the procedures provided in this section, refer to Chapter 6 of the *JavaTest Architect's Guide* for additional information regarding creating configuration interviews.

1. **Define the environment variable required by the test.**

   In the example (`Test2.java`), the test case `SampleStringValue()` checks if the `sampleValue` that is passed in equals `Hello`.

   The harness requires the `sampleValue` environment variable to run the test. Because the value for `sampleValue` cannot be known ahead of time, it must be provided in the configuration interview by the user.

   The `decodeArg(String[],int)`method decodes the argument and passes the value to the `sampleValue` environment variable. The harness uses the `-stringValue` argument in accordance with the `executeArgs` argument entry in the test description file.

2. **Specify the environment value using the test description entry.**

   The test description file (`index.html`) can use either the `executeArgs` or `context` to identify the environment variables required by the test. In our example, the environment variable is called `sample.string.value`.

   The following is an example of using the `executeArgs` argument (also see `SimpleTestSuite/tests/sample/pkg2/index.html`).

```
<tr>
<td scope="row"> <b>executeArgs</b> </td>
<td> -stringValue $sample.string.value </td>
</tr>
```

   As an alternative, you can use the `context` field. If you use the `context` field in the test description, the argument in the `decodeArgs(String,int)` method must be changed accordingly with `-sample.string.value` used as the argument.

   The following is an example of using the `context` field.

```
<tr>
<td scope="row"> <b>context</b> </td>
<td> sample.string.value </td>
</tr>
```

3. **Write the configuration interview class.**

   The Framework provides a basic configuration interview that defines parameters common for all Java ME technology test suites. If your test suite requires additional parameters, you must create a sub-interview and link it to the Framework interview.

   In the `SampleInterview` example, the test requires an additional parameter. The following steps describe how to create the sub-interview that adds the additional parameter to the configuration.

   a. **Create a class (**`SampleInterview`**) that is a subtype of the** `Interview` **class.**

      The following code creates the `SampleInterview` class.

```
public class SampleInterview extends Interview {
....
}
```

b. **Identify the new interview (`SampleInterview`) as a sub-interview of the parent interview.**

   In the new sub-interview class, the constructor must take a reference to the parent interview as an argument and pass this reference to the superclass constructor. This identifies the interview as a sub-interview of the parent interview.

   The following code identifies `SampleInterview` as a sub-interview of `MidpTckBaseInterview`.

```
public class SampleInterview extends Interview {

   public SampleInterview(MidpTckBaseInterview parent)
             throws Fault {
        super(parent, "sample");
}
```

c. **(Optional) Identify the resource file and helpset file used by the sub-interview.**

   By default, a sub-interview shares a resource file and More Info help files with its parent interview. However, you can choose to use a different resource file and helpset file.

   The following code in the `SampleInterview` example specifies a different resource file and helpset file.

```
public class SampleInterview extends Interview {
   public SampleInterview(MidpTckBaseInterview parent)
            throws Fault {
        super(parent, "sample");
        setResourceBundle("i18n");
        setHelpSet("help/sampleInterview");
   }
}
```

   In the example, `"i18n"` is the properties file updated in Step 4, and `sampleInterview` is the More Info helpset file updated in Step 5.

d. **Use the `setFirstQuestion` method in the constructor to specify the first question in the sub-interview.**

   The following `setFirstQuestion` code in the `SampleInterview` example specifies the first question.

```
public class SampleInterview extends Interview {
   public SampleInterview(MidpTckBaseInterview parent)
            throws Fault {
        super(parent, "sample");
        setResourceBundle("i18n");
        setHelpSet("help/sampleInterview");
        setFirstQuestion(first);
    }
}
```

e. **Specify the question type.**

   The `Question` class is a base class that provides the different types of questions required to build the sub-interview. In the example, the question gets string information. Therefore, the example must use the `StringQuestion` type.

   The following code in the `SampleInterview` example specifies the `StringQuestion` type.

```
StringQuestion first = new StringQuestion(this, "hello")
```

   In the example, `hello` is a unique id that identifies the specific question name. This name is used in the properties and map files.

f. **Implement exporting the configuration value to the test environment.**

   One of the goals of the interview is to export the configuration value to the test environment. Each question has an `export()` method that is used for this purpose.

   The following code in the `SampleInterview` example exports the value to the test environment.

```
StringQuestion first = new StringQuestion(this, "hello") {
   public void export(Map map) {
      map.put("sample.string.value", String.valueOf(value));
   }
}
```

   An alternative is to use the `setExporter()` method to export the value. The following code is an example of using the `setExporter()` method.

```
first = new StringQuestion(this, "hello");
first.setExporter(
Exporters.getStringValueExporter("sample.string.value"));
```

g. **Implement error checking for the question answer.**

   If the user provides an invalid answer to a question, the interview cannot proceed. For most questions, error conditions are handled by returning null, which causes the Configuration Editor to display an invalid response message in red at the bottom of the question pane. Alternatively, if the Framework's interview extension library is used, it's possible to implement validation without subclassing the question via `Question.setValidator()`. For detailed information, see the `com.sun.tck.j2me.interview.lib` package API documentation.

The following code in the `SampleInterview` example implements error checking.

```
StringQuestion first = new StringQuestion(this, "hello") {
   public boolean isValueValid() {
      return value != null && value != "";
   }

   public void export(Map map) {
      map.put("sample.string.value", String.valueOf(value));
   }
}
```

h. **Use the `getNext()` method to determine the next question in the sub-interview.**

   Every question except the Final question must provide a `getNext()` method that determines the next (successor) question or null. Alternatively, if the Framework's interview extension library is used, it's possible to link questions without subclassing via `Question.setPathResolver()` or `Question.linkTo()` methods. For detailed information, see the `com.sun.tck.j2me.interview.lib` package API documentation.

   The following code in the `SampleInterview` example specifies the next configuration question.

```
Question qXXX = ......... {
   Question getNext() {
      return qNextQuestion;
   }
};
```

i. **Repeat Step e through Step h until all configuration questions are added to the sub-interview.**

j. Use the FinalQuestion marker to identify the last question in the sub-interview.

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

The following code in the `SampleInterview` example identifies the final question in the sub-interview.

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

The following `SampleInterview.java` class is used for the example in the Simple Test Suite. It puts everything together that was described in Step a through Step j.

```
public class SampleInterview extends Interview {
   public SampleInterview(MidpTckBaseInterview parent) throws Fault {
      super(parent, "sample");
      setResourceBundle("i18n");
      setHelpSet("help/sampleInterview");
      first = new StringQuestion(this, "hello");
      first.setExporter(
      Exporters.getStringValueExporter("sample.string.value"));
      first.linkTo(end);
      setFirstQuestion(firstQuestion);
   }

   private StringQuestion first;
   private final FinalQuestion end = new FinalQuestion(this, "end");
}
```

You can also view the `SampleInterview.java` file in the following location:

`SimpleTestSuite/src/sample/suite`

4. **Update the interview `.properties` (resource) file.**

   All question text is located in the interview `.properties` file associated with the interview class files and is identified by a unique question key.

   The question key is based on a name assigned by the test developer and must uniquely identify the question with the interview. Question keys are created in the following form:

   ```
   interview-class-name.question-name
   ```

   The interview `.properties` file contains the following types of elements:

   - The title of the full interview
   - A title for each question

     Question titles take the following form:

     question-key`.smry` = *title-text*
   - The text for each question

     Question text takes the following form:

     question-key`.text` = *question-text*
   - Additional entries for choice items that are localized

   For every interview question that you create, you must add corresponding `.smry` and `.text` entries in the interview `.properties` file. You can either update the existing file or create a new one.

   In the example, the interview class is named `SampleInterview` and the question name is `hello`. For the example, the following entries must be in the interview `.properties` file.

   ```
   SampleInterview.hello.smry = Sample Config Property
   SampleInterview.hello.text = Enter "Hello" here and the ^
   test will verify the value.
   ```

   To view the complete contents of the file, see the i18n.properties file in the following Simple Test Suite source location:

   ```
   SimpleTestSuite/src/sample/suite
   ```

5. **Set up the More Info system.**

   The JavaHelp™ system libraries required to display More Info in the Configuration Editor are included in the `javatest.jar`. However, you must configure the More Info system to include corresponding More Info topics for the questions in the interview. The following steps describe how to set up the More Info system.

a. **Create a `help` directory under the directory where the interview classes are located.**

   For this example, use the following location:

   `SimpleTestSuite/src/sample/suite/help`

b. **Create a helpset file (`sampleInterview.hs`) under the `help` directory.**

   The helpset file specifies the location of the map file for the More Info system. The following example shows the contents of the `sampleInterview.hs` helpset file used in the Simple Test Suite.

```
<!DOCTYPE helpset PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp HelpSet
Version 1.0//EN" "http://java.sun.com/products/javahelp/helpset_1_0.dtd">
<helpset version="1.0">
<!-- title -->
<title>Simple Test Suite Configuration Interview - Help</title>
<!-- maps -->
<maps>
<mapref location="default/sampleInterview.jhm"/>
</maps>
</helpset>
```

   In the preceding helpset example, `sampleInterview.jhm` is the map file specified for the More Info system.

   You can also view the sampleInterview.hs file in the following location:

   `SimpleTestSuite/src/sample/suite/help`

c. **Create a `default` directory under the `help` directory.**

   For the SampleTestSuite example, use the following location:

   `SimpleTestSuite/src/sample/suite/help/default`

d. **Create a map file (`sampleInterview.jhm`) in the `default` directory.**

The JavaHelp system uses IDs from a map file to identify both the location and the HTML files that it loads for the More Info system. Each More Info file must have a corresponding entry in the map file of the form `<mapID target=”`*name*`” url=”location/filename.html”/>`.

The following example shows the contents of the map file used in the Simple Test Suite.

```
<!DOCTYPE map PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map
Version 1.0//EN" "http://java.sun.com/products/javahelp/map_1_0.dtd">

<map version="1.0">

<!-- SampleInterview -->
<mapID target="SampleInterview.hello"
url="SampleInterview/sampleStringValue.html"/>
</map>
```

The `target` attribute defines a unique ID for the topic file. The `url` attribute defines the path to and name of the HTML topic file (relative to the map file). The `.html` files in the map file example are the More Info files.

You can also view the sampleInterview.jhm map file in the following location:

`SimpleTestSuite/src/sample/suite/help/default`

e. **Create the More Info topic files.**

The More Info topic files are provided in HTML format and displayed in the Configuration Editor's More Info pane. Refer to the *JavaTest Architect's Guide* for the procedure for creating More Info topic files. The following is an example of a More Info file.

```
<html>
<head>
<title>
SampleInterview.hello
</title>
<LINK REL="stylesheet" TYPE="text/css" HREF="../wizard.css" TITLE=
"Style">
</head>
<body>
The test suite architect can provide "More Info" here, explaining
the question in more detail.
</body>
</html>
```

Additional examples of More Info files can be found in the following directory.

```
SimpleTestSuite/src/sample/suite/help/default/SampleInterview
```

6. **Create a JAR file containing the configuration interview.**

   After creating the interview, you must package it into a JAR file for inclusion with the test suite. If you include other custom components with your test suite, they can be packaged with the interview. This is usually done in the build.

   If you successfully ran the build as described in Chapter 3, "Building an Updated Simple Test Suite" on page 23, an example of this JAR file is in the `lib` directory. In the example, the JAR file is named `SimpleTestSuite-build/lib/sample_jt.jar`.

7. **Add the JAR file to the classpath entry of the `testsuite.jtt` file.**

See Also

Before you can use a custom interview to run tests, it must be plugged into the JavaTest harness. See "To Plug In a Custom Interview" on page 44.

To build a test suite containing the configuration interview and associated tests, see "Building a Test Suite" on page 45.

Plugging in a Custom Interview

Before you can use a custom interview to run tests it must be correctly plugged into the JavaTest harness.

▼ To Plug In a Custom Interview

This procedure describes how to plug a custom `Interview` class into the JavaTest harness.

Before plugging a custom `Interview` class into the JavaTest harness, you must create a custom `TestSuite` class (`com.sun.javatest.TestSuite` should be among its parents).

1. **Set the `classpath` entry in the `testsuite.jtt` file to specify the location of the test suite class.**

   The `testsuite.jtt` file is a main marker or configuration file for every JavaTest harness based test suite (including all ME Framework test suites). It resides in the root of the test suite. In the `testsuite.jtt` file there is a testsuite entry that specifies the test suite class used to construct everything else.

   The following example is the `classpath` entry in the `testsuite.jtt` file used by the Simple Test Suite (see "`testsuite.jtt` File" on page 28):

   ```
   classpath=lib/j2mefw_jt.jar lib/sample_jt.jar lib/interviewlib.jar
   ```

2. **Set the `testsuite` entry in the `testsuite.jtt` file to specify the name of the test suite class.**

   The following example is the `testsuite` entry in the `testsuite.jtt` file used by the Simple Test Suite (see "`testsuite.jtt` File" on page 28):

   ```
   testsuite=sample.suite.SampleTestSuite
   ```

3. **Modify the source for your test suite class to specify which interview to use.**

   Override the `createInterview()` method (inherited from the base class, `com.sun.javatest.TestSuite`) and specify your interview class.

   An alternative approach to specifying the interview would be to include an `interview` entry in the `testsuite.jtt` file. However, not all test suites look into this entry. Some test suites might use the `createInterview()` method and not look into the `testsuite.jtt` file for an `interview` entry.

Additional Action

Now, you have your `testsuite.jtt` file, TestSuite class, and Interview class plugged into the JavaTest harness.

The only thing that remains is how to write the interview for ME Framework based test suites, which classes to extend, etc.

Building a Test Suite

Refer to Chapter 3, "Building an Updated Simple Test Suite" on page 23 for the procedure to build a Java ME technology test suite by using Framework resources.

Note – If using Linux to build a test suite, refer to the system requirements section of the appropriate WTK documentation for required configuration information.

CHAPTER 5

Writing Tests That Use Framework Resources

This chapter describes how to write different types of Java ME technology tests that use Java ME TCK Framework resources and to set special properties required by the tests. General information about the process of creating tests can be located in Chapters 5 and 6 of the *Java Technology Test Suite Development Guide*. Additional information about writing tests for use with the JavaTest harness can be located in Chapters 3 of the *JavaTest Architect's Guide*.

This chapter contains the following sections:

Testing Devices With Automated Tests

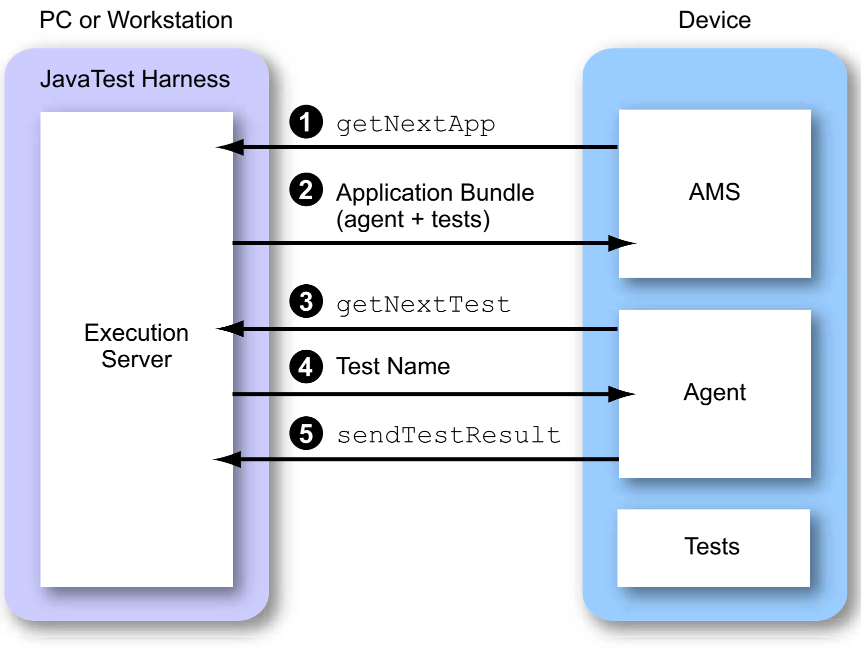
Automated tests are the most common and the most simple of the tests that are run on the device. In this configuration, the harness (running on a PC or a workstation) sends an application bundle containing the tests and an agent to the device where they are unpacked by the Application Management Software (AMS) built into the device and run. In this configuration, user interaction is not required to run each test.

See Chapter 3 for a detailed description, procedure, and example of writing automated tests. In addition, `SimplestTestSuite`, `SimpleTestSuite`, and `AdvancedTestSuite` directories contain examples of automated tests.

Automated Test Execution

FIGURE 5-1 and the associated text are for CLDC and MIDP execution mode. In the diagram, arrows indicate the direction of dataflow between the device and the workstation. The numbered items indicate the sequence and the content of the dataflow.

FIGURE 5-1 Automated Test Execution



1. `getNextApp` - The AMS issues a `getNextApp` command to the execution server on the workstation.

   The AMS implementation is device specific and must be provided by the licensee.

2. **Application Bundle** - The execution server sends an application bundle to the AMS.

   The AMS downloads and executes the application bundle on the device. Test bundles are created by the Framework.

3. `getNextTest` - The agent issues a `getNextTest` request to the execution server.
4. **Test Name** - The execution server returns the name of the next test in the application bundle that the agent should run.

   While the tests from the application bundle are loaded onto the device by the AMS, the execution server establishes the sequence in which the tests in the bundle are run.
5. `sendTestResult` - The agent returns the test results to the execution server.

Items 3, 4, and 5 repeat until all tests in the application bundle are run. When all tests in the application bundle are run, the AMS requests the next application bundle (items 1 and 2) and the sequence is repeated until all tests and all test bundles in the test suite are run.

Testing Communications or Networking With Distributed Tests

Distributed tests are a special type of automated tests. Not only do they have a device side test component, which is executed under the control of a test agent (as with any regular automated tests), but they also have one or more remote components on other devices or the Java SE platform side. The distributed test components have names and communicate with each other by sending messages to each other by way of a messaging service. The remote components of a distributed test typically run on a harness host by using a passive agent in the same virtual machine as the harness and provide some additional functionality needed by the test. For example, a test verifies that an HTTPS connection can be made to the remote host. The remote component that runs on the Java SE platform would be an HTTPS server.

To develop distributed tests, the test writer must write classes for both the client and remote test components as well as create an appropriate test description file. The `SimpleTestSuite` directory contains examples of distributed tests.

Client Test Component

In the MIDP case, the client test component of the distributed test must extend the `com.sun.tck.j2me.services.messagingService.J2MEDistributedTest` class. In the CDC case, the client test component of the distributed test must extend the `com.sun.tck.j2me.services.messagingService.CDCDistributedTest` class.

These are the base classes for distributed tests for the Java ME technology component of the test. Each component of the distributed test has a unique name used for identification during message exchange.

The client test component must use the `send` and `handleMessage` methods in the `J2MEDistributedTest` class to send and receive messages to or from the other named components of the distributed test.

Remote Test Component

The remote test component of the distributed test must extend the `com.sun.tck.j2me.services.messagingService.J2SEDistributedTest` class and, to send messages to the client component of the distributed test, invoke the `send` method in the `DistributedTest`.

Test Description for Distributed Tests

The test description file for the distributed test must contain the `distributed` keyword and the `remote` attribute. The test description might also include additional attributes such as the `remoteSource` and `executeArgs` attributes.

Required Distributed Test Keyword

The `distributed` keyword identifies the type of test to the harness and enables test selection by the user. If more than one keyword is specified, the names must be separated by white space. The following is an example of the `distributed` keyword entry that must be added to the test description file.

CODE EXAMPLE 5-1 Required Distributed Test Keyword

```
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD> distributed </TD>
</TR>
```

`remote` Attribute

The `remote` attribute contains the execution command for the remote test components of the distributed test group. The values for these attributes are exported from the configuration interview and defined in the test environment file.

The following is an example of the `remote` attribute entry that must be added to the test description file.

CODE EXAMPLE 5-2 `remote` Attribute

```
<TR>
<TD> <B>remote</B></TD>
<TD>  networkAgent: sample.pkg.SampleDistributedTest
-msgSwitch $testMsgSwitch</TD>
</TR>
```

In the example, the `remote` attribute execution command is composed of the following values:

- `sample.pkg.SampleDistributedTest` is the fully qualified name of the remote test class.
- `$testMsgSwitch` is an environment variable used by the Framework to start the remote test component.

`remoteSource` Attribute

The `remoteSource` attribute contains the name of the remote test class of the distributed test group. If more than one remote test source file is specified in the test description, the names must be separated by white space.

The following is an example of the `remoteSource` attribute entry in which the remote test class is `SampleDistributedTest.java`.

CODE EXAMPLE 5-3 `remoteSource` Attribute

```
<TR>
<TD> <B>remoteSource</B> </TD>
<TD> SampleDistributedTest.java </TD>
</TR>
```

`executeArgs` Attribute

The `executeArgs` attribute contains the environment variables that are passed to the test classes or client test component being executed. The values for these attributes are exported from the configuration interview and defined in the test environment. When more than one environment variable is specified in the test description, the names and values must be separated by white space.

The following is an example of the executeArgs attribute entry in which the $testMsgSwitch and $timeOut environment variables are specified.

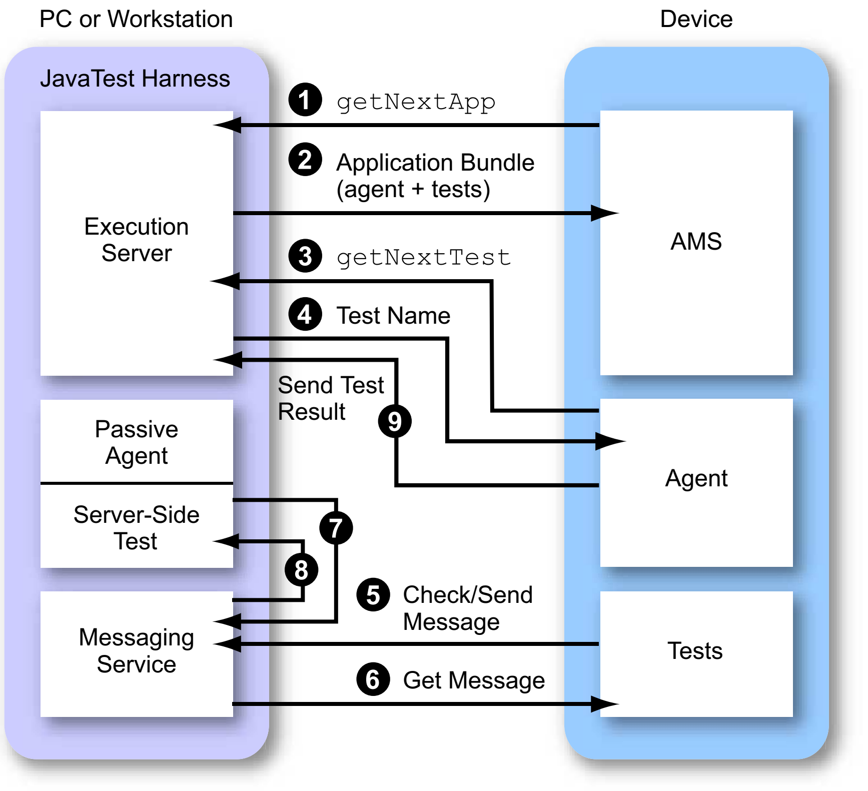
CODE EXAMPLE 5-4 executeArgs Attribute With Multiple Environment Variables

```
<TR>
<TD> <B>executeArgs</B>
<TD>  -msgSwitch $testMsgSwitch -timeout $timeOut
</TR>
```

Distributed Test Execution

FIGURE 5-2 and the associated text are for MIDP execution mode. In the diagram, arrows indicate the direction of dataflow between the device and the workstation. The numbered items indicate the sequence and the content of the dataflow.

FIGURE 5-2 Distributed Test Execution



1. `getNextApp` - The AMS issues a `getNextApp` command to the execution server on the workstation.

   The AMS implementation is device specific and must be provided by the licensee.

2. **Application Bundle** - The execution server sends an application bundle to the AMS.

   The AMS downloads and executes the application bundle on the device. Test bundles are created by the Framework.

3. `getNextTest` - The agent issues a `getNextTest` request to the execution server.

4. **Test Name** - The execution server returns the name of the next test in the application bundle that the AMS should run.

   While the tests from the application bundle are loaded onto the device by the AMS, the execution server establishes the sequence in which the tests in the bundle are run.

5. **Check or Send Message**- The test sends either a message or a check message request to the Framework messaging service on the harness.

6. **Get Message** - The Framework messaging service on the harness sends the test a get message command.

7. **Check or Send Message** - The server-side test sends either a message or a check message request to the Framework messaging service on the harness.

8. **Get Message** - The Framework messaging service on the harness sends the server-side test a get message command.

9. `sendTestResult` - The agent returns the test result to the execution server.

The agent repeats items 3, 4, and 9 until all tests in the application bundle are run. When all tests in the application bundle are run, the AMS requests the next application bundle (items 1 and 2) and the sequence is repeated until all tests and all test bundles in the test suite are run.

Testing User Interfaces With Interactive Tests

Interactive tests are the tests that require some form of user interaction and cannot be executed without such interaction. From a design point of view, interactive tests are a subtype of distributed test. As a subtype of distributed test, interactive tests generally execute on the test device under the control of a component called an agent. However, unlike distributed tests, interactive tests also require some form of user interaction as a part of the test. Interactive tests might require that the user change something with a device, which triggers event generation or require the user to verify that a device plays sound or vibrates. But most typically, interactive tests are used to validate user interface on the device.

The Framework supports the following three types of interactive modes:

- **Yes-No** - The user must determine the outcome of the test.
- **Done** - The user must perform specific actions and confirm the completion, but cannot affect the outcome of the test.
- **Information-only** - The visual component is presented to the users, but no user action is required (this is the default mode).

To develop interactive tests, the test writer must write classes for both the client and the remote test components of the interactive test as well as create an appropriate test description file. The `AdvancedTestSuite` directory contains examples of interactive tests.

Client Test Component

The client test component of the interactive test must extend the `MidDistribInteractiveTest` class. This is the base class for distributed interactive tests and extends the `com.sun.tck.j2me.services.messagingService.J2MEDistributedTest` class. The `MidDistribInteractiveTest` class provides all of the common interfaces required by distributed interactive tests (such as `setFormTitle`, `waitForMessage`, `messageReceived`, `addStatus`, and `cleanUp`).

The client test component must call the `send` and `handleMessage` methods in the `MidDistribInteractiveTest` class to send and receive messages to or from the other components of the interactive test group.

The following directory contains an example of a client test component (`MidDistribInteractiveTest.java`):

`AdvancedTestSuite/src/share/classes/com/sun/tck/midp/lib/`

Remote Test Component

The remote test component of the interactive test must extend the `com.sun.tck.midp.lib.DistribInteractiveTest` class. The `DistribInteractiveTest` class is the base class for the Java Platform, Standard Edition (Java SE platform) technology component of the interactive test. It is a subclass of the J2SEDistributedTest class that requires user action before a test result is determined. The remote test component must invoke the send method to exchange messages from the client test components.

A remote test component usually contains the information that is displayed by the harness (such as the test instruction, expected image, or the pass-fail messages for the test cases).

Test Description for Interactive Tests

Like distributed tests, the test description file for the interactive test must contain the distributed keyword and the remote attribute. In addition, interactive test description files must also contain the interactive keyword.

See "remote Attribute" on page 50 for a description and example of the remote attribute.

The test description might also include the remoteSource and executeArgs attributes. See "remoteSource Attribute" on page 51 and "executeArgs Attribute" on page 51 for a description and example of these attribute entries.

Required Interactive Test Keywords

The following is an example of the keywords that must be added to the test description file for an interactive test. Both the distributed keyword and the interactive keyword are required. When additional keywords are specified, the names must be separated by white space.

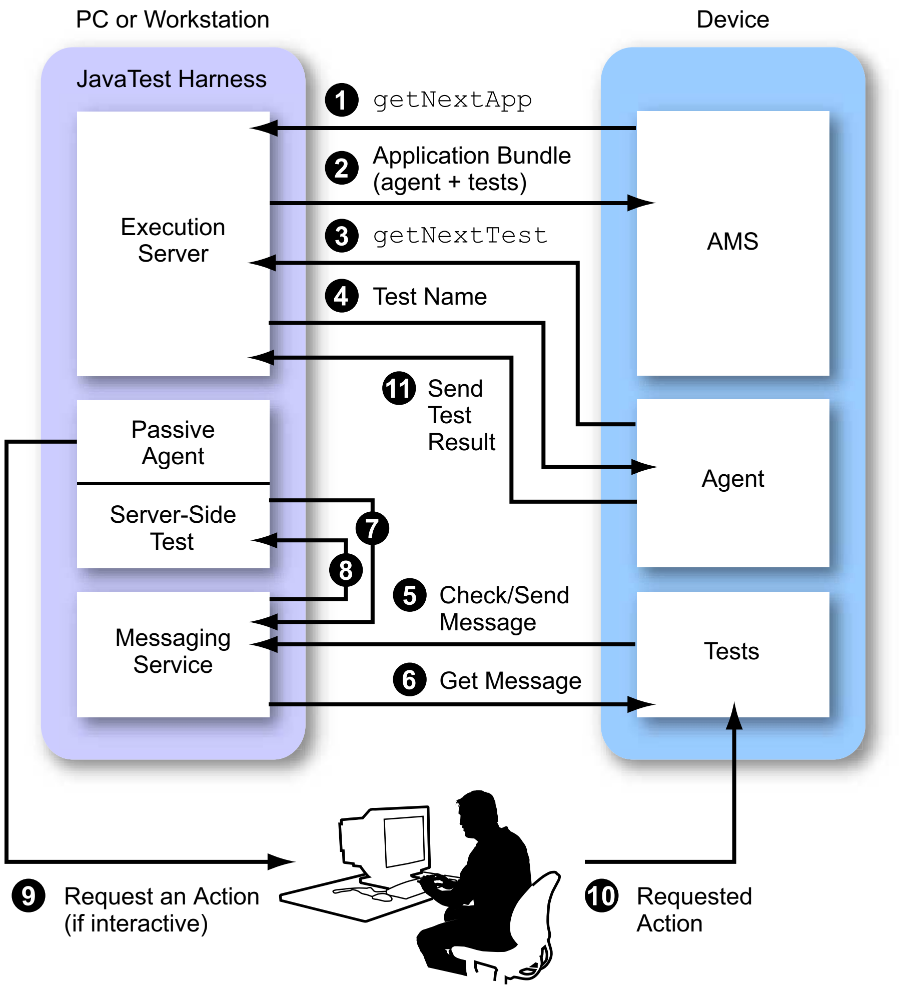
CODE EXAMPLE 5-5 Required Interactive Test Keywords

```
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD> distributed interactive </TD>
</TR>
```

Interactive Test Execution

FIGURE 5-3 and the associated text are for MIDP execution mode. In the diagram, arrows indicate the direction of dataflow between the device, the workstation, and the tester. The numbered items indicate the sequence and content of the dataflow.

FIGURE 5-3 Interactive Test Execution



1. `getNextApp` - The AMS issues a `getNextApp` command to the execution server on the workstation.

   The AMS implementation is device specific and must be provided by the licensee.

2. **Application Bundle** - The execution server sends an application bundle to the AMS.

   The AMS downloads and executes the application bundle on the device. Test bundles are created by the Framework.

3. `getNextTest` - The agent issues a `getNextTest` request to the execution server.

4. **Test Name** - The execution server returns the name of the next test in the application bundle that the AMS should run.

   While the tests from the application bundle are loaded onto the device by the AMS, the execution server establishes the sequence in which the tests in the bundle are run.

5. **Check or Send Message** - The test sends either a message or a check message request to the Framework messaging service on the harness.

6. **Get Message** - The Framework messaging service on the harness sends the test a get message command.

7. **Check or Send Message** - The server-side test sends either a message or a check message request to the Framework messaging service on the harness.

8. **Get Message** - The Framework messaging service on the harness sends the server-side test a get message command.

9. **Request an Action**- The Framework messaging service sends a message request to the user that requires an action on a test.

10. **Action** - The user performs the action on the test requested by the messaging service.

11. `sendTestResult` - The agent returns the test results to the harness.

The agent repeats items 3, 4, and 5 until all tests in the application bundle are run. When all tests in the application bundle are run, the AMS requests the next application bundle (items 1 and 2) and the sequence is repeated until all tests and all test bundles in the test suite are run.

Example of an Interactive Test

Examples of interactive tests can be viewed (and executed) in the `AdvancedTestSuite` directory of the Framework bundle.

Testing Application Provisioning With OTA Tests

Over-the-air (OTA) tests are MIDP-specific tests that verify an OTA application provisioning implementation. This includes obtaining, installing, and removing applications (MIDlet suites), and enforcing security requirements. Each OTA test has an associated application (a MIDlet suite) that is downloaded from the provisioning server and is installed and launched on the test device.

When developing an OTA test, a test writer must write classes for both the server and client test components as well as create an appropriate test description file.

Server Component of an OTA Test

The server component of the OTA tests must extend the `com.sun.tck.midp.lib.OTATest` class. This is the base class for OTA tests and provides convenience methods that install, execute, and remove applications called MIDlet suites.

There are different kinds of OTA tests with each kind of OTA test having one or more of the following functional requirements:

- Requires the successful installation of the MIDlet suite
- Requires that the MIDlet suite installation fail
- Executes the MIDlet suite
- Does not execute the MIDlet suite

After the execution of the OTA test, the installed MIDlet suite must be removed. The server component of the typical OTA test invokes the `install`, `run` and `remove` methods to install, execute, and remove the MIDlet suite. See "Example of OTA Test" on page 65 for an example of a server component.

Client Test Component of an OTA Test

The client test component, called the MIDlet suite, contains the actual test case. The MIDlet suite must be in the format of a JAR file or a Java Application Descriptor (JAD) file so that it can be downloaded from the provisioning server and installed and launched on the test device. The creation of the JAR file or JAD file is usually done at the build time. If the test writer creates a JAR file or a JAD file with special information about the MIDlet suite as part of the bundle, the test writer must also create a manifest file (.mf).

Test Description for OTA Tests

The test description file for OTA tests must contain the ota keyword, the executeClass attribute, and the remote attribute. The test description might also include other attributes such as the remoteSource and executeArgs attributes. See "OTA Test Description Examples" on page 62 for examples of complete test description files.

Required OTA Test Keyword

The ota keyword identifies the type of test to the harness and enables test selection by the user. If more than one keyword is specified, the names must be separated by white space. The following is an example of the ota keyword entry that must be added to the test description file.

CODE EXAMPLE 5-6 Required OTA Test Keyword

```
<TR>
<TD> <B>keywords</B></TD>
<TD> ota </TD>
</TR>
```

executeClass Attribute

The executeClass attribute specifies the client test class, while the remote attribute specifies the execution command for the server test component. The following is an example of the executeClass attribute entry that must be added to the test description file.

CODE EXAMPLE 5-7 executeClass Attribute Entry

```
<TR>
<TD> <B>executeClass</B></TD>
<TD> sample.pkg.OTA.Test_MIDlet</TD>
</TR>
```

remote Attribute

The remote attribute specifies the execution command for the server test component. The following is an example of the remote attribute entry that must be added to the test description file.

CODE EXAMPLE 5-8 remote Attribute Entry

```
<TR>
<TD> <B>remote</B></TD>
<TD> networkAgent: sample.pkg.OTA.OTATest1 -httpServerPort
$httpServerPort -testDir $testDir -OTAHandlerClass
$OTAHandlerClass
-OTAHandlerArgs $OTAHandlerArgs</TD>
</TR>
```

In the remote attribute, sample.pkg.OTA.OTATest1 is the test class of the server component and all other arguments (-httpServerPort $httpServerPort -testDir $testDir -OTAHandlerClass $OTAHandlerClass -OTAHandlerArgs $OTAHandlerArgs) are required for OTA tests.

If the test is written for a trusted MIDlet, the argument, -signer=$jks.signer "-signerArgs=$jks.signer.args", must also be included in the remote attribute. This argument is used by the Framework to sign the MIDlet.

The following is an example of the remote attribute entry added to the test description file for an OTA test that is written for a trusted MIDlet.

CODE EXAMPLE 5-9 remote Attribute Entry for Trusted MIDlet

```
<TR>
<TD> <B>remote</B></TD>
<TD> networkAgent: sample.pkg.OTA.OTATest1 -httpServerPort
$httpServerPort -testDir $testDir -OTAHandlerClass
$OTAHandlerClass
-OTAHandlerArgs $OTAHandlerArgs -signer=$jks.signer
"-signerArgs=$jks.signer.args"</TD>
<TR>
```

OTA Test Description Examples

The following is an example of a basic OTA test description file.

CODE EXAMPLE 5-10 OTA Test Description File

```
<TITLE>Test Specifications and Descriptions for Test</TITLE>
</HEAD>
<BODY>
<H1>Test Specifications and Descriptions for Test</H1>
<HR>
<a name="Test"></a>
<TABLE BORDER=1 CLASS=TestDescription>
<TR>
<TD> <B>title</B></TD>
<TD> Over-the-Air Test</TD>
</TR>

<TR>
<TD> <B>source</B></TD>
<TD> <A HREF=
"midlet/Test_MIDlet.java">midlet/Test_MIDlet.java</A></TD>
</TR>
<TR>
<TD> <B>executeClass</B></TD>
<TD> sample.pkg.OTA.Test_MIDlet</TD>
</TR>
<TR>
<TD> <B>keywords</B></TD>
<TD> runtime positive ota </TD>
<TR>
<TD> <B>remote</B></TD>
<TD> networkAgent: sample.pkg.OTA.OTATest1 -httpServerPort
$httpServerPort -testDir $testDir -OTAHandlerClass
$OTAHandlerClass
-OTAHandlerArgs $OTAHandlerArgs</TD>
<TR>
<TD> <B>remoteSource</B></TD>
<TD> <A HREF="OTATest1.java">OTATest1.java</TD>
</TABLE>

</BODY>
</HTML>
```

The following is an example of a test description for a trusted OTA test. See "remote Attribute" on page 61 for a description of the values that must be set in the remote attribute for a trusted OTA test.

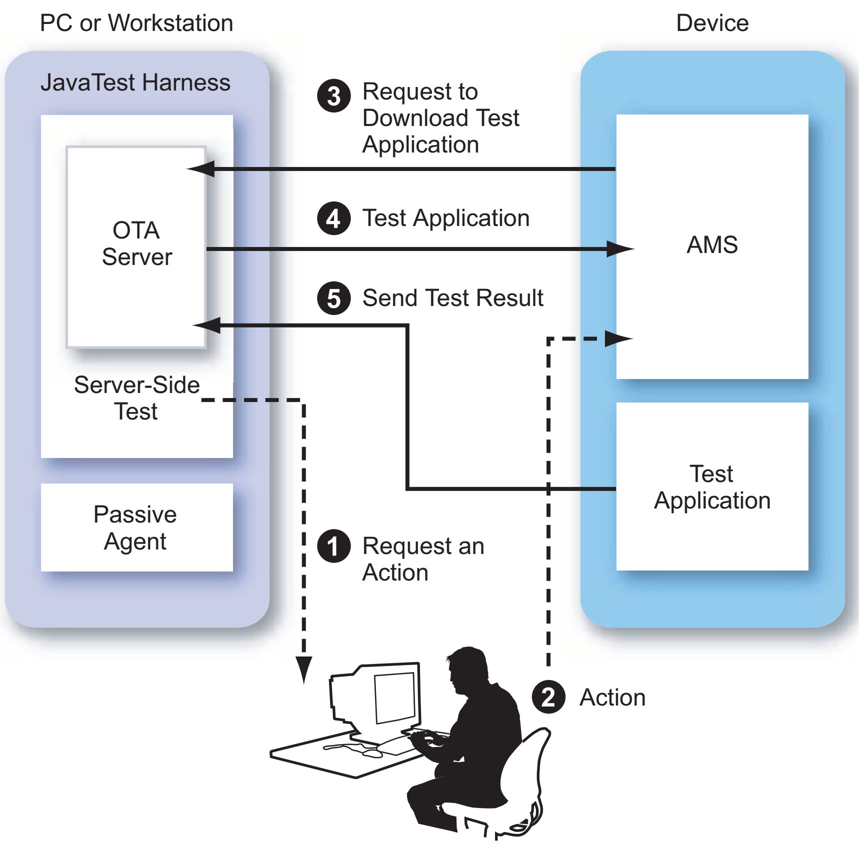
CODE EXAMPLE 5-11 Trusted OTA Test Description File

```
<TITLE>Test Specifications and Descriptions for Test</TITLE>
</HEAD>
<BODY>
<H1>Test Specifications and Descriptions for Test</H1>
<HR>
<a name="Test"></a>
<TABLE BORDER=1 CLASS=TestDescription>
<TR>
<TD> <B>title</B></TD>
<TD> Over-the-Air Test for trusted midlet</TD>
</TR>

<TR>
<TD> <B>source</B></TD>
<TD> <A HREF=
"midlet/Test_MIDlet.java">midlet/Test_MIDlet.java</A></TD>
</TR>
<TR>
<TD> <B>executeClass</B></TD>
<TD> sample.pkg.OTA.Test_MIDlet</TD>
</TR>
<TR>
<TD> <B>keywords</B></TD>
<TD> runtime positive trusted ota </TD>
<TR>
<TD> <B>remote</B></TD>
<TD> networkAgent: sample.pkg.OTA.OTATest1 -httpServerPort
$httpServerPort -testDir $testDir -OTAHandlerClass
$OTAHandlerClass
-OTAHandlerArgs $OTAHandlerArgs -signer=$jks.signer
"-signerArgs=$jks.signer.args"</TD>
<TR>
<TD> <B>remoteSource</B></TD>
<TD> <A HREF="OTATest1.java">OTATest1.java</TD>
</TABLE>

</BODY>
</HTML>
```

OTA Test Execution

In FIGURE 5-4, arrows indicate the direction of dataflow between the device and the workstation. The numbered items indicate the sequence and the content of the dataflow.

FIGURE 5-4 OTA Test Execution



1. **Request an Action** - The server-side test requests an action from the test operator.

2. **Action** - The test operator action is sent to the AMS.

3. **Request to Download Test Application** - The AMS sends the OTA server a request for a download of a test application.

4. **Test Application** - The OTA server sends the AMS a test application.

The AMS downloads and executes the test application on the device.

5. **Send Test Result** - The test application sends the test results to the OTA server.

Example of OTA Test

The following is an example of an OTA server component. In the example, the source of the MIDlet suite is named `Test_MIDlet.java` and the JAD file is `Test_MIDlet.jad`. The JAD file contains the manifest file and the `Test_MIDlet.class`.

CODE EXAMPLE 5-12 Server Test Component Example

```
public class OTATest1 extends OTATest {

   public static final String NAME = "Test_MIDlet";
   public static final String testCaseID = "OTA_Sample";


   public Status OTA_Sample() {
      Attributes attr = new Attributes();

      attr.putValue("reportURL", address + "/TestResult/");
      attr.putValue("testCaseID", testCaseID);

      String jadURL = address + "/" + NAME + ".jad";

      log.println("Installing " + NAME + ".jad\n");

      Status s = installWithNotification(attr, NAME, null);
      if (s == null || !s.isPassed()) {
          log.println("Could not install the application.");
          return s;
      }

      try {
          log.println("Running " + NAME + ".jar" + "\n");
          String[] cmt = new String[]{"Some comments"};

          String qn = "Some Questions";

          s = handler.run(jadURL, NAME, cmt, qn);

          if (s == null || !s.isPassed()) {
              log.println("Could not run midlet suite");
              return s;
          }
```

CODE EXAMPLE 5-12 Server Test Component Example *(Continued)*

```
            return otaServer.getStatus(testCaseID);
          } finally {
              if (!handler.removeAll(null).isPassed()) {
                  ref.println("Could not remove midlet suites");
              }
          }
      }

  }
```

Testing Security-Constrained Functionality With Security Tests

This section addresses writing different types of security tests, using keywords and attributes to mark different types of security tests, and using attributes in the test description to grant or deny the security permissions for the tests.

Types of Security Tests

Security related tests can be grouped into the following types:

- Untrusted
- Trusted
- Double-duty

Untrusted Tests

Untrusted tests verify API implementation behavior for unsigned MIDlet Suites. Untrusted tests verify that the assertions related to untrusted MIDlet suites are properly implemented.

All untrusted MIDlet suites can run in a restricted environment where access to protected APIs or functions is either not allowed, or allowed only with explicit user permission. Untrusted tests must not be run in trusted security mode. See "Marking Untrusted Tests" on page 67 for a description of how to use keywords to mark untrusted tests.

Trusted Tests

Trusted tests verify API implementation behavior for signed MIDlet suites. In most cases, these tests verify that specification assertions related to signed MIDlet suites are properly implemented.

Trusted MIDlet suites can be permitted to access APIs that are considered sensitive or to which access is restricted without any user action. The required permissions are granted without any explicit user action. Trusted tests must not be run in untrusted security mode. See "Marking Trusted Tests" on page 69 for a description of how to use keywords to mark trusted tests.

Double-duty Tests

Double-duty tests verify API implementation behavior that depends on security factors. For example, tests for a security sensitive API that require specific permissions to be granted or denied. Double-duty tests must be run in both the trusted and untrusted security mode. See "Using an Attribute to Mark Double-Duty Tests" on page 69 for a description of how to use the `DoubleDutySecurity` attribute to mark double-duty tests.

Using Keywords to Mark Security Tests

When developing security tests, tests writer should use an appropriate keyword in the test description to mark the type of test. The keyword enables users to select or exclude tests from a test run based on the security mode.

Marking Untrusted Tests

When developing untrusted tests, the test writers should include the `untrusted` keyword in the test description. With the `untrusted` keyword included in the test description, the untrusted test is selected and executed during a test run in untrusted security mode. Tests marked with the `trusted` keyword are not selected and executed in the untrusted security mode.

The following is an example of an `untrusted` keyword entry added to a test description file.

CODE EXAMPLE 5-13 `untrusted` Keyword Entry in the Test Description

```
<TR>
```

CODE EXAMPLE 5-13 `untrusted` Keyword Entry in the Test Description *(Continued)*

```
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD>untrusted</TD>
</TR>
```

If other keywords (such as `positive`) are used, they would be included in the same line as the `untrusted` keyword. The following is an example of a test description file that uses `runtime`, `positive`, and `untrusted` keywords.

CODE EXAMPLE 5-14 Test Description for an `untrusted` Test

```
<TITLE>Test Specifications and Descriptions for Test</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Test</H1>
<HR>

<a name="Test"></a>
<TABLE BORDER=1 SUMMARY="JavaTest Test Description" CLASS=
TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH SCOPE=
"col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> checking untrusted test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test.java">Test.java</A> </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg.Test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD>runtime positive untrusted</TD>
</TR>
</TABLE>

</BODY>
</HTML
```

Marking Trusted Tests

When developing trusted tests, the test writers should use the `trusted` keyword in the test description to mark these tests. With the `trusted` keyword, all the trusted tests are selected and executed during the trusted security mode. Tests that have the `untrusted` keyword are not selected and executed in the trusted security mode.

The following is an example of adding a `trusted` keyword entry to a test description file.

CODE EXAMPLE 5-15 `trusted` Keyword Entry

```
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD>trusted</TD>
</TR>
```

When other keywords (such as `positive`) are used, they are included in the same line as the `trusted` keyword and separated by white space.

Using an Attribute to Mark Double-Duty Tests

When developing double-duty tests, test writers should use the `DoubleDutySecurity` attribute with the value set to `yes` in the test description to mark these tests. Tests that have a `DoubleDutySecurity` attribute with a value of `yes` in their test description file are selected and executed in both the trusted and the untrusted security modes.

The following is an example of a `DoubleDutySecurity` attribute with a value of `yes` added to a test description file.

CODE EXAMPLE 5-16 `DoubleDutySecurity` Attribute

```
<TR>
<TD SCOPE="row"> <B>DoubleDutySecurity</B> </TD>
<TD> yes </TD>
</TR>
```

The following is an example of a test description file for a double-duty test.

CODE EXAMPLE 5-17 Test Description for a Double Duty Test

```
<TITLE>Test Specifications and Descriptions for Test</TITLE>
</HEAD>

<BODY>
```

CODE EXAMPLE 5-17 Test Description for a Double Duty Test *(Continued)*

```
<H1>Test Specifications and Descriptions for Test</H1>
<HR>

<a name="Test"></a>
<TABLE BORDER=1 SUMMARY="JavaTest Test Description" CLASS=
TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH SCOPE=
"col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> checking double duty test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test.java">Test.java</A> </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg.Test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>DoubleDutySecurity</B> </TD>
<TD> yes </TD>
</TR>
</TABLE>

</BODY>
</HTML
```

Granting or Denying Security Permissions

A security test might require that certain permissions be granted (or denied) for the test to pass. The test is run or not run according to the permissions that a test writer grants or denies.

Granting Security Permissions

When writing security tests, a test writer can specify the permissions that the security policy must grant in the protection domain for the test application to execute and pass. A test writer can specify the required security permissions by including a `grant` attribute in the test description. If the security policy doesn't grant the specified permissions, the test must be filtered out of the test run.

For example, a test application is written based on the assumption that following permissions are granted:

- javax.microedition.io.Connector.file.read
- javax.microedition.io.Connector.file.write

The following is an example of a `grant` attribute and permissions added to a test description file.

CODE EXAMPLE 5-18 `grant` Attribute Entry and Security Permissions

```
<TR>
<TD SCOPE="row"> <B>grant</B> </TD>
<TD> javax.microedition.io.Connector.file.read
javax.microedition.io.Connector.file.write </TD>
</TR>
```

The value of the `grant` attribute is a list of space-separated permissions (in the example, `javax.microedition.io.Connector.file.read` and `javax.microedition.io.Connector.file.write`) that must be granted by the security policy for this test application to execute and pass. If these permissions are not granted for the test application, the test application must be filtered out.

The following is an example of a test description file that includes the `grant` attribute and permissions.

CODE EXAMPLE 5-19 Test Description That Grants Permissions for a Security Test

```
<TITLE>Test Specifications and Descriptions for Test</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Test</H1>
<HR>

<a name="Test"></a>
<TABLE BORDER=1 SUMMARY="Javatest Test Description" CLASS=
TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH SCOPE=
"col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> checking grant permission</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test.java">Test.java</A> </TD>
</TR>
<TR>
```

CODE EXAMPLE 5-19 Test Description That Grants Permissions for a Security Test

```
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg.Test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>grant</B> </TD>
<TD> javax.microedition.io.Connector.file.read
javax.microedition.io.Connector.file.write </TD>
</TR>

</TABLE>
</BODY>
</HTML>
```

Denying Security Permissions

When writing security tests, a test writer can specify the permissions that the security policy must not grant in the protection domain for this test application to execute and pass. Test writers can specify the denied security permissions by including a `deny` attribute in the test description. If the security policy grants the specified permissions, the test must be filtered out of the test run.

For example, suppose a test application is written to expect that a security exception is thrown because the `javax.microedition.io.Connector.file.read` permission is not granted. If the security policy grants the `javax.microedition.io.Connector.file.read` permission to the test application, the test must be filtered out and not run.

The following is an example of the `deny` attribute and permission added to a test description file.

CODE EXAMPLE 5-20 `deny` Attribute in the Test Description

```
<TR>
<TD SCOPE="row"> <B>deny</B> </TD>
<TD> javax.microedition.io.Connector.file.read </TD>
</TR>
```

The value of the `deny` attribute is a list of space-separated permissions that must be denied by the security policy for this test application to execute and pass. If the permissions are granted in the protection domain for the test application, the test application must be filtered out and not run.

The following is an example of a test description file that includes the `deny` attribute and permissions.

CODE EXAMPLE 5-21 Test Description That Denies Permissions for a Security Test

```
<TITLE>Test Specifications and Descriptions for Test</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Test</H1>
<HR>

<a name="Test"></a>
<TABLE BORDER=1 SUMMARY="Javatest Test Description" CLASS=
TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH SCOPE=
"col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> checking deny permission</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test.java">Test.java</A> </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg.Test</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>deny</B> </TD>
<TD> javax.microedition.io.Connector.file.read </TD>
</TR>

</TABLE>
</BODY>
</HTML>
```

Adding Resource Files in Tests

Test writers must sometimes develop tests that require extra resource files (such as image, data, or class files) for the test execution. When writing tests for CLDC and MIDP-based implementations, test writers can use the `resources` entry in the test description file to specify the location of the resource files.

During test execution, the test execution framework bundles the resource files (specified in the test description) and the test class files (listed in the `testClasses.lst`) into a test JAR file.

For example, the following test requires a `Duke.png` to create the Image object for successful execution.

CODE EXAMPLE 5-22 Test That Requires an Image Resource

```
Public class Test2 {

    public Status testImage() {

        Image imgI = null;
        String imageDir = "/shared/sample/pkg3/";
        try {
            imgI = Image.createImage(imageDir+"Duke.png");
        } catch (IOException e) {
            ....
        }
        if (imgI == null) {
            return Status.failed("Failed: no image is created.");
        }
        return Status.passed("OKAY");
    }
}
```

For this test example, the following is a `resources` entry that might be added to the test description file.

CODE EXAMPLE 5-23 `resources` Attribute in the Test Description

```
<TR>
<TD SCOPE="row"> <B>resources</B> </TD>
<TD>shared/sample/pkg3/Duke.png</TD>
</TR>
```

In the `resources` entry, the value of the resource (`shared/sample/pkg3/Duke.png`) is the qualified name of the resource file. If multiple resources are added to the test description file, separate them with white space. If the resource file is a class file, the file name must include the `.class` extension.

During build time, the `Duke.png` image file is copied into the specified destination directory (`../shared/sample/pkg3/Duke.png`). During test execution, the execution framework checks the `testClasses.lst` file and the test URL. During the test execution, the framework bundles the resource files specified in the `resource` entry (`../shared/sample/pkg3/Duke.png`) and the class files listed in `testClasses.lst` into a test JAR file.

The following is an example of a test description file containing a `resources` entry, directory, and file name.

CODE EXAMPLE 5-24 Test Description That Includes Resources

```
<HTML><HEAD>
<TITLE>Test Specifications and Descriptions for Test2</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Test2</H1>
<HR>

<a name="Test2"></a>
<TABLE BORDER=1 SUMMARY="Javatest Test Description" CLASS=
TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH
SCOPE="col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> Checking image creation</TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test2.java">Test2.java</A> </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg3.Test2 </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD>runtime positive </TD>
<TR>
<TD SCOPE="row"> <B>resources</B> </TD>
<TD>shared/sample/pkg3/Duke.png</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

Enabling Test Selection

Test selection in a test run is of value to a user when a test suite includes tests for optional features that were not implemented. Because the assumption of tests is that the target implementation must support these features, the implementation fails tests that are not be applicable to it. This section describes how a developer can enable the filtering of the tests (test selection) in a test suite.

Factors and Mechanisms for Test Selection

The following factors affect how tests are selected for a test run and describe several mechanisms that exist for users to enable test selection.

- **Keywords** - A standard test selection mechanism provided by the harness that enables a user to filter tests based on specific descriptive keywords assigned by a developer to a test.

  This mechanism is convenient for logical test grouping. The following are examples of test filtering using keywords in the test description file.

  - Type (distributed, interactive, or OTA)

    See "Required Distributed Test Keyword" on page 50, "Required Interactive Test Keywords" on page 56, and "Required OTA Test Keyword" on page 60.

  - Security mode (trusted or untrusted)

    See "Using Keywords to Mark Security Tests" on page 67.

  - Custom keywords specified by the user

    See Appendix C for a list of the Framework keywords.

- **Prior test status** - A built-in mechanism in the harness that enables a user to filter tests based on the previous runs.

  The test developer is not required to perform any action to enable this filtering mechanism. Filtering is normally set by the user through the Prior Status question in the standard configuration interview.

- **Exclude list** - Excludes certain test cases from the certification test run.

  The exclude list (`testsuite.jtx`) file identifies the tests in a test suite that should not be run. The exclude list is located in the test suite `lib` directory. Test developers use the following format to add tests to the exclude list:

  Test-URL[*Test-Cases*] *BugID Keyword*

  See Chapter 4 for additional information about the exclude list.

- `selectIf` **expression in test description** - This mechanism provides a flexible way for developers to make individual tests selectable (`com.sun.tck.j2me.javatest.ExprFilter`) by including a `selectIf` expression in test description.

  The test developer can include a `selectIf` expression in the test description file for a test. Each `selectIf` field contains a Boolean expression that is evaluated by the filter. The test is selected by the harness for a test run if the value is true. See "`selectIf` Test Selection" on page 77.

- **Grant or deny mechanism** - This mechanism enables users to select tests based on the security requirements (`com.sun.tck.midp.policy.PermissionFilter`) specified by the test developer.

  Tests are selected if all the permissions listed by the developer in the grant or deny test description field are granted or denied. See "Granting or Denying Security Permissions" on page 70.

- **Custom test suite-specific filters** - Developers can extend the test suite with custom test suite-specific filters. The *JavaTest Architect's Guide* describes creating custom filters.

`selectIf` Test Selection

The following procedures describe how developers can use the `selectIf` expression to enable filtering tests that are not applicable for an implementation.

▼ To Enable Test Selection with the `selectIF` Expression

The `selectIf` entry in the test description file contains a Boolean expression which is evaluated by a test filter. If the Boolean expression evaluates to false, the test is not run. If the expression evaluates to true, the test is run.

1. **Add a `selectIf` entry to the test description file for a test class that users might be required to filter out of a test run.**

   The following is an example of a `selectIF` entry added to a test description file.

```
<TR>
<TD SCOPE="row"> <B>selectIf</B> </TD>
<TD> isFeatureSupported </TD>
</TR>
```

   In the example, `isFeatureSupported` is an environment variable. Most environment variables can be used as Boolean expressions to filter out tests from a test run. Usually, the value of an environment variable is target implementation specific and must be provided by the user through the configuration interview.

   Step 2 describes the procedure for writing an interview question that collects this information from the user.

   The following example is a complete test description file that includes a `selectIf` entry.

```
<TITLE>Test Specifications and Descriptions</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Test</H1>
<HR>

<a name="Test"></a>
<TABLE BORDER=1 SUMMARY="JavaTest Test Description" CLASS=
TestDescription>
<THEAD><TR><TH SCOPE="col">Item</TH><TH SCOPE=
"col">Value</TH></TR></THEAD>
<TR>
<TD SCOPE="row"> <B>title</B> </TD>
<TD> Checking constructors </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>source</B> </TD>
<TD> <A HREF="Test.java">Test.java</A> </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>executeClass</B> </TD>
<TD> sample.pkg.Test </TD>
</TR>
<TR>
<TD SCOPE="row"> <B>keywords</B> </TD>
<TD>runtime positive </TD>
</TR>
<TR>
```

```
<TD SCOPE="row"> <B>selectIf</B> </TD>
<TD> isFeatureSupported </TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

2. **Write an interview question that obtains the value of the environment variable from the user.**

   To obtain the value of environment variables from users, test developers must write an interview class and create the required interview questions. If an interview class already exists in the test source, the developer can either add new questions to the existing interview or create a new sub-interview class containing the required questions and link it to the existing interview class.

   For this example, we will assume that we have an existing interview class (`SampleInterview.java`) in the Simple Test Suite source and that we are adding a new question about whether the target implementation supports a specific feature.

   For an example of linking a new interview class to an existing interview, see Chapter 4 "Creating and Using a Configuration Interview" on page 32.

   The following is an example of question code that can be added to an interview.

```
// The added question:Does the implementation support the feature?
private YesNoQuestion qFeatureSupport = new YesNoQuestion(this,
"featureSupport") {
   public void clear() {
      setValue(YesNoQuestion.NO);
   }

protected void export(Map data) {
   boolean b = (value == YES);
   data.put("isFeatureSupported", String.valueOf(b));
}
```

   In the question code, the `isFeatureSupported` environment variable name must be consistent with the name used in Step 1 for `selectIf` (in the test description file). Also in the question code, `featureSupport` is the unique question name that becomes part of the question key created in Step 3 for use in both the map and the properties files.

   The following example is a complete configuration interview (`SampleInterview.java`) that contains the added question code.

```
public class SampleInterview extends Interview {
   /**
   * @param parent
   * @throws Fault
   */
   public SampleInterview(MidpTckBaseInterview parent)
            throws Fault {
      super(parent, "sample");
      init();
   }
   private void init() throws Fault {
        setResourceBundle("i18n");
        setHelpSet("help/sampleInterview");

        first = new StringQuestion(this, "hello");
        first.setExporter(
           Exporters.getStringValueExporter("sample.string.value"));
        first.linkTo(qFeatureSupport);
        setFirstQuestion(first);
    }

// This is the added question: Does the implementation support the feature ?
    private YesNoQuestion qFeatureSupport = new YesNoQuestion(this,
              "featureSupport") {
        public void clear() {
           setValue(YesNoQuestion.NO);
        }

        protected void export(Map data) {
           boolean b = (value == YES);
           data.put("isFeatureSupported", String.valueOf(b));
        }

        protected Question getNext() {
           return end;
        }
    };

    private StringQuestion first;
    private final FinalQuestion end = new FinalQuestion(this, "end");

}
```

3. **Add interview question `.smry` and `.text` entries to the `.properties` file.**

   For each added interview question, the developer must create corresponding `.smry` and `.text` entries in the resource file (`.properties`). The following is an example of the `.smry` and `.text` entries added for the new question.

```
SampleInterview.featureSupport.smry = Feature Support
SampleInterview.featureSupport.text = Does your system support the feature ... ?
```

   The `.smry` and `.text` entries contain elements used to perform the following functions:

   - `SampleInterview.featureSupport` is a unique question key that the Configuration Editor uses to identify the required question and its corresponding More Info topic.

     Question keys are created in the following form:

interview-class-name.question-name

- The `.smry` entry specifies the question title (in the example, `Feature Support`) that the Configuration Editor displays to the user.
- The `.text` entry specifies the question text (in the example, `Does your system support the feature that ... ?`) that the Configuration Editor displays to the user.

4. **Create a More Info topic file for the question.**

   The More Info system of the Configuration Editor displays topic files for each question. Each file provides detailed information that the user needs when answering its associated interview question. Refer to Chapter 6 in the *JavaTest Architect's Guide* for additional information about creating More Info topic files.

   Examples of More Info topic files can be found in the following `SampleTestSuite` location.

```
AdvancedTestSuite/sampletck/src/sample/suite/help/default/SampleInterview/
```

5. **Update the map file.**

   After the topic file is created, the map file must be updated.

   If the topic file for the new question is named `featureSupport.html`, the test developer must add the following line to the `sampleInterview.jhm` file.

```
<mapID target="SampleInterview.featureSupported"
url="SampleInterview/featureSupport.html"/>
```

Note – The `sampleInterview.jhm` map file is located in the `AdvancedTestSuite/sampletck/src/sample/suite/help/default/` directory.

---

6. **Create the JAR file.**

   After creating the interview, you must package it into a JAR file for inclusion with the test suite during the build time.

   If you successfully ran the build as described in Chapter 3, "Building an Updated Simple Test Suite" on page 23, you have an example of this JAR file under the `lib` directory. In the example, the JAR file is named `AdvancedTestSuite/lib/sample_jt.jar`.

7. **Add the JAR file to the classpath entry of the `testsuite.jtt` file.**

Additional Action

The test suite containing the configuration interview and associated tests can now be built. See "Building a Test Suite" on page 45, in Chapter 4.

CHAPTER **6**

Using the ME Framework Agent

This chapter describes how to use the ME Framework Agent (agent). The agent is a small Java technology application used in conjunction with the harness to run tests on a Java platform on which it is not possible or desirable to run the main harness. The agent runs on the test platform and responds to requests from the harness, which runs on a separate platform.

The ME Framework 1.2.1 contains a rewritten agent subsystem that does not use the standard JavaTest harness Agent Monitor. It has a different set of command-line parameters from those described in the JavaTest harness documentation as well as a different set of JAR files and application main classes when compared to those in Framework version 1.1.2.

Starting the Agent

The agent resides in the `main_agent.jar` file and is started from a command line. When starting an agent, the communication type specified for the agent must correspond to the communication type specified in the configuration interview. If you specify a communication type for the agent that differs from the type set in the configuration interview, you must change the communication type in the configuration interview to correspond with the agent before running tests.

The Framework provides built-in support for socket (TCP/IP), datagram (UDP), HTTP, and serial (RS-232) communication types. Custom communication types provided by users can also be specified.

Using TCP/IP Communication

The simplest configuration for starting an agent is to specify a connection over TCP/IP with the default port number unchanged in the configuration interview. The following is an example of the command used to start the agent that uses the TCP/IP communication type:

```
java -cp main_agent.jar com.sun.tck.j2me.agent.AgentMain -tcp \
          -activeHost JTHARNESS-HOST-NAME -trace
```

In the example, the following settings and options are used:

- *JTHARNESS-HOST-NAME* represents the name of the host on which the JavaTest harness is running.
- The `-trace` option displays the test progress and any errors that are reported.
- The `-tcp` option specifies that the agent use the socket communication type (TCP/IP).

See "Displaying Agent Command Line Parameters" on page 86 for additional agent parameters and functionality

Using UDP Communication

To start an agent that uses the datagram communication type (UDP), replace the `-tcp` option in the previous example with `-udp`.

Note – To run tests with the agent, you must also change the communication type in the configuration interview to UDP.

The following is an example of the command used to start the agent that uses the datagram communication type:

```
java -cp main_agent.jar com.sun.tck.j2me.agent.AgentMain -udp \
          -activeHost JTHARNESS-HOST-NAME -trace
```

See "Displaying Agent Command Line Parameters" on page 86 for additional agent parameters and functionality.

Using HTTP Communication

A more complex configuration for starting an agent is to specify a connection HTTP. To start agent that uses the HTTP communication, additional options are required in the command.

Note – To run tests with the agent, you must also change the communication type in the configuration interview to HTTP.

The following is an example of the command used to start an agent that uses the HTTP communication type:

```
java -cp main_agent.jar:midp_commClient.jar \
          com.sun.tck.j2me.agent.AgentMain \
          -serverAddress \
          http://JTHARNESS-HOST-NAME:JTHARNESS-PORT \
          -commImplClass \
          com.sun.tck.j2me.services.commService.clients.MidHTTPCommClient
          -trace
```

See "Displaying Agent Command Line Parameters" on page 86 for additional agent parameters and functionality.

Using Serial Communication

Serial communication between the harness and the agent requires use of the the Comm API (`http://java.sun.com/products/javacomm/`) on the JavaTest harness side and Generic Connection Framework from Java ME (`javax.microedition.io` package) on the agent side.

To enable serial support for communication between JavaTest harness and agent make sure the Comm API is properly configured and the harness is started with Comm API supported. Also make sure the implementation supports the `comm: protocol` in the context of `javax.microedition.io`.

To start an agent with serial communication, use the `GenericCommClient`. The connection string must be constructed as defined in the `CommConnection` class of CDC version 1.1. Parity and stop bits are configured through the string baud rate. Default values are implementation dependent.

`GenericCommClient` has two optional parameters, `-limitPacketSize` and `-verbose`. These parameters can be provided as `-commArgs` arguments. The value of `-limitPacketSize` limits the number of bytes sent at one time by the device to harness side. The default value is 4096 bytes.

The `-verbose` option is very useful for troubleshooting problems with serial connection. In this mode, all communication between the agent and the JavaTest harness is displayed in the console.

The following is an example of the command used to start an agent that uses the serial communication type:

```
java -cp main_agent.jar:generic_commClient.jar \
          com.sun.tck.j2me.agent.AgentMain \
          -trace \
          -serverAddress "comm:com0;baudrate=19200" \
          -commImplClass \
          com.sun.tck.j2me.services.commService.clients.GenericCommClient\
          -commArgs -verbose
```

See "Displaying Agent Command Line Parameters" on page 86 for additional agent parameters and functionality.

Displaying Agent Command Line Parameters

Use the following command to display the command line parameters:

```
java -cp main_agent.jar com.sun.tck.j2me.agent.AgentMain -help
```

The command prints information about the agent command line parameters. TABLE 6-1 lists and describes the agent command line parameters.

TABLE 6-1 ME Agent Command Line Parameters

Parameter	**Description**
`-activeHost` *host*	Specifies the host name for the standard TCP or UDP clients. `localhost` is the default value.
`-activePort` *port*	Specifies the port number for the standard TCP or UDP clients. `8188` is the default value.
`-commArgs` *arg1 ... argN*	Specifies the `CommunicationClient` initialization parameters. They must be the last parameters in the command line.
`-commArgsStart` *arg1 ... argN* `-commArgsEnd`	Specifies the `CommunicationClient` initialization parameters.
`-commImplClass` *class-name*	Specifies the name of the `CommunicationClient` implementation.
`-concurrency` *number*	Sets the maximum number of simultaneous connections.
`-dumpSystemProps`	Adds system properties to the registration parameters.

TABLE 6-1 ME Agent Command Line Parameters *(Continued)*

Parameter	Description
`-help`	Prints a list and description of command line parameters.
`-map` *file-name*	Specifies a map file for translating the arguments of incoming requests.
`-P`*key=value*	Defines the registration parameter.
`-serverAddress` *address*	Specifies the server address.
`-tcp`	Specifies using of the standard TCP/IP client.
`-testClassesURL`	Pointer to test classes in URL format. Can refer to a remote source. Only a single URL can be specified for each parameter. Use multiple `-testClassesURL` parameters to specify multiple sources for test classes.
`-trace`	Sets tracing of the agent execution.
`-udp`	Specifies using of the standard UDP client.
`-usage`	Prints a list and description of command line parameters.

APPENDIX **A**

Test API

The following is the Java ME technology-specific API that every test developer must know:

- `Test`
- `Status`
- `MultiTest`
- `J2MEDistributedTest`
- `J2SEDistributedTest`
- `DistribInteractiveTest`
- `OTATest`

`Test`

Interface name: `com.sun.tck.cldc.lib.Test`

This is a Java ME technology version of the standard harness Test interface. This interface is implemented by all Java ME technology tests. Each test must define the `run` method as follows:

CODE EXAMPLE A-1 `run` Method

```
public Status run(String[] args, PrintStream log, PrintStream ref)
```

A test must also define `main` as follows:

CODE EXAMPLE A-2 Definition of main

```
public static void main(String[] args) {
Test t = new <test-class-name>();
Status s = t.run(args, System.err, System.out);
s.exit();
}
```

Defining `main` in this manner enables the test to also be run standalone, independent of the harness.

Status

Class name: `com.sun.tck.cldc.lib.Status`

This is a Java ME technology version of the standard harness `Status` class. It embodies the result of a test, a status-code, and a related message.

MultiTest

Class name: `com.sun.tck.cldc.lib.MultiTest`

This is a Java ME technology version of the standard harness `MultiTest` class. It serves as a base class for tests with multiple sub test cases. This base class implements the standard `com.sun.tck.cldc.lib.Test` features so that you can provide the additional test cases with little concern about the boilerplate needed to execute individual test case methods (such as update the `runTestCases()` method to add check and invocation statement). `MultiTest` is designed as a base class used during development of new test classes.

You must add individual test case methods to your derived test class to create a useful test class. Each test case method must take no arguments. If you need to pass an argument into a test method, design a wrapper test case to calculate the argument values and then call the test method with the correct arguments. The test case methods must follow the convention shown in CODE EXAMPLE A-3.

CODE EXAMPLE A-3 Test Case Method

```
public Status methodName()
```

J2MEDistributedTest

Class name:
`com.sun.tck.j2me.services.messagingService.J2MEDistributedTest`

This is the base class for the Java ME technology component of the distributed test. Each distributed test has a unique name used for identification during message exchange. The `send` and `handleMessage` methods can be used to send and receive messages to or from the other named components of the test.

CDCDistributedTest

Class name:
`com.sun.tck.j2me.services.messagingService.CDCDistributedTest`

This is the base class for the Java ME technology component of a distributed test that targets CDC environments. Each distributed test has a unique name used for identification during message exchange. The `send` and `handleMessage` methods can be used to send and receive messages to or from other named components of the test.

J2SEDistributedTest

Class name:
`com.sun.tck.j2me.services.messagingService.J2SEDistributedTest`

This is the base class for the Java SE technology component of the distributed test. This is a Java SE technology counterpart for `J2MEDistributedTest`.

DistribInteractiveTest

Class name: `com.sun.tck.midp.lib.DistribInteractiveTest`

This is the base class for the Java SE technology component of the interactive test. It is a subclass of `J2SEDistributedTest` that requires user action before test result is determined. Three different user interface types are supported:

- **Yes-No** - The user is required to determine the outcome of the test.
- **Done** - The user is required to perform certain actions and confirm the completion, but cannot otherwise affect the outcome of the test.
- **Information only** - A visual component is presented to the user, but no user action is required.

OTATest

Class name: `com.sun.tck.midp.lib.OTATest`

This is the base class for the Java SE technology component of an OTA test. It provides various convenience methods which trigger installation, execution, or removal of an application.

APPENDIX **B**

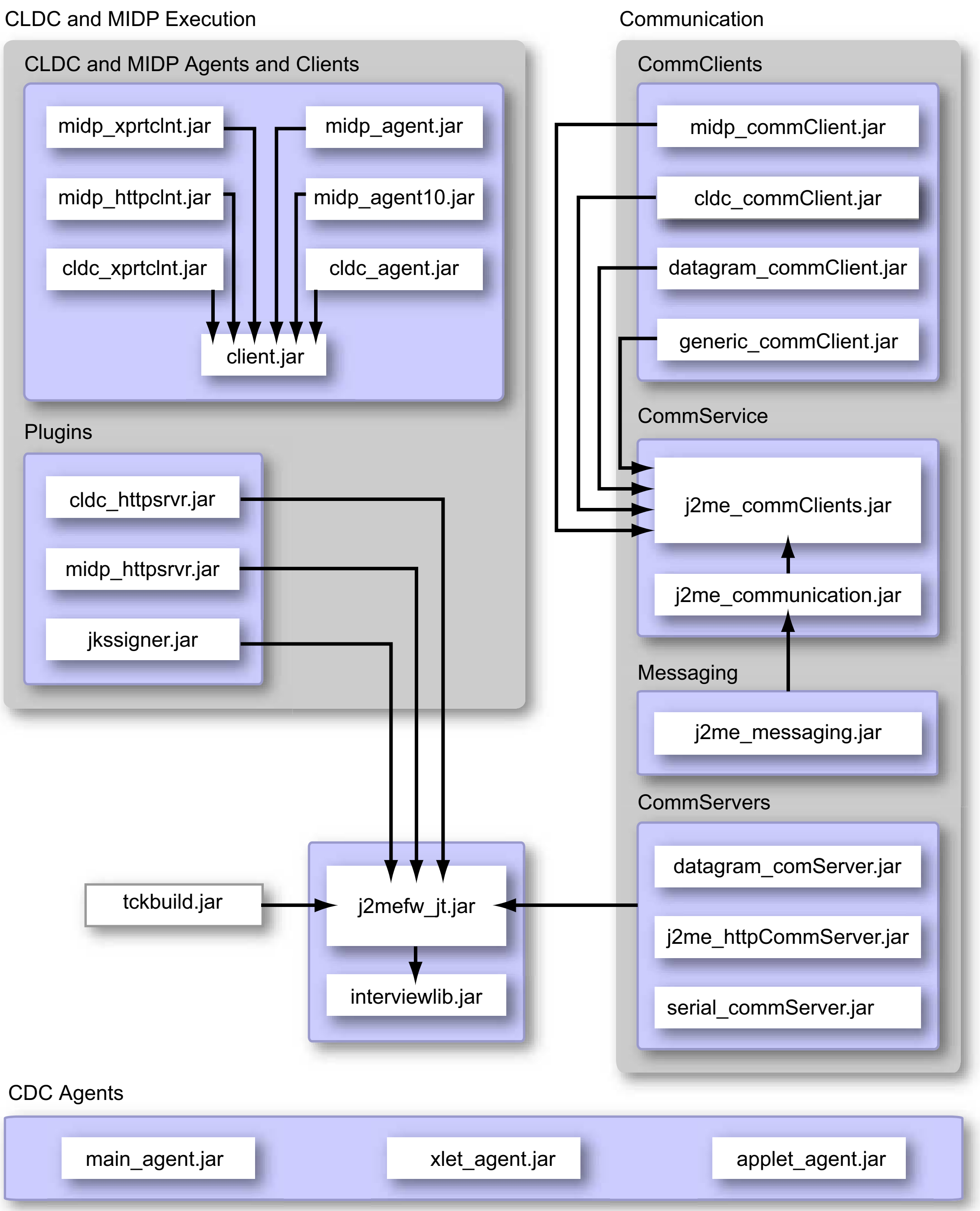
Framework Redistributables Directory

This appendix describes the contents of the redistributables directory provided by the Java ME TCK Framework bundle. The contents of the redistributables directory are organized in the following directory structure:

- `lib` Directory
- `src` Directory
- `doc` Directory

`lib` Directory

The `lib` directory contains the Java ME TCK Framework JAR files and keystore files. The files provided in the `lib` directory are not part of the Java ME TCK Framework functional groupings. The following figure illustrates the dependencies between the JAR files in this directory.


CLDC and MIDP Execution
CLDC and MIDP Agents and Clients
midp_xprtclnt.jar
midp_agent.jar
midp_httpclnt.jar
midp_agent10.jar
cldc_xprtclnt.jar
cldc_agent.jar
client.jar
Plugins
cldc_httpsrvr.jar
midp_httpsrvr.jar
jkssigner.jar
Communication
CommClients
midp_commClient.jar
cldc_commClient.jar
datagram_commClient.jar
generic_commClient.jar
CommService
j2me_commClients.jar
j2me_communication.jar
Messaging
j2me_messaging.jar
CommServers
datagram_comServer.jar
j2me_httpCommServer.jar
serial_commServer.jar
tckbuild.jar
j2mefw_jt.jar
interviewlib.jar
CDC Agents
main_agent.jar
xlet_agent.jar
applet_agent.jar

- `tckbuild.jar` - Utility classes used by TCK build.
- `j2me-tck-framework_121.txt` - Java ME TCK Framework version marker (this file has no content).
- `midptck.ks` - Standard keystore containing default certificates used for MIDP application signing.

  The keystore file is shipped as part of the Framework. The following information is required to utilize it:

  - Certificate Alias: `dummyCA`
  - Keystore Password: `keystorepwd`
  - Private Key Password: `keypwd`

  The certificate is already integrated in the WTK.

Core

The following are core Java ME TCK Framework files provided in the `lib` directory:

- `j2mefw_jt.jar` - Harness plug-in code.

  Most of the Java SE platform code to support ME TCKs is in this file.
- `interviewlib.jar` - Helper library for interview creation.

CLDC and MIDP Execution

The Java ME TCK Framework provides the following two functional groupings of CLDC and MIDP execution files:

- Agent and client `.jar` files
- Plug-in `.jar` files

CLDC and MIDP Agents and Clients

The following are CLDC and MIDP agent and client `.jar` files provided in the `lib` directory:

- `midp_xprtclnt.jar` - Client for use in Test Export mode.

  This client does not implement a communication channel with a remote machine.
- `midp_httpclnt.jar` - HTTP-based communication client for MIDP.
- `cldc_httpclnt.jar` - HTTP-based communication client for CLDC.
- `client.jar` - Client interface.

This is the primary communication interface for the pluggable communication protocol used by CLDC-based agents.

- `midp_agent.jar` - Standard harness agent for MIDP 2.x.

  This agent works in conjunction with an HTTP-based communication client (`midp_httpclnt.jar`)

  Application model: MIDlet

  Communication channel: HTTP
- `midp_agent10.jar` - Standard harness agent for MIDP 1.0.

  This agent works in conjunction with an HTTP-based communication client (`midp_httpclnt.jar`). This agent does not support the MIDP 2.x security model.

  Application model: MIDlet

  Communication channel: HTTP
- `cldc_agent.jar` - Standard harness agent for CLDC.

  Application model: Main

  Communication channel: pluggable.

Plug-ins

The following are the plug-in `.jar` files provided in the `lib` directory:

- `cldc_httpsrvr.jar` - Implementation of the server interface for the harness plug-in.

  This server works with the HTTP-based Client for CLDC.
- `midp_httpsrvr.jar` - Implementation of the server interface for the harness plug-in.

  This server works with the HTTP-based Client for MIDP.
- `jkssigner.jar` - X.509 signer for MIDP application JAR files.

Communication

The Java ME TCK Framework provides four functional groups of `.jar` files for use in communications between agents and the test harness as well as for use with the distibuted test framework (DTF). The Messaging group of `.jar` files is used for the distributed testing framework (DTF), while the CommService, CommClients, and CommServers functional groups are for use in communication between agent and test harness as well as the DTF.

Messaging

`j2me_messaging.jar` provides the Java ME technology messaging service classes. These classes are the base classes for distributed tests.

CommService

The following are the CommService `.jar` files provided in the `lib` directory:

- `j2me_commClients.jar` - Distributed test communication client interface.
- `j2me_communication.jar` - Service that provides communication between the harness and the remote device.

CommClients

The following are the CommClients `.jar` files provided in the `lib` directory:

- `midp_commClient.jar` - Communication client interface implementation that uses HTTP for communication.
- `cldc_commClient.jar` - Implementation of the Client interface for the CLDC device.

  This client uses HTTP for communication.
- `datagram_commClient.jar` - Communication client interface implementation that uses datagrams for communication.
- `generic_commClient.jar` - Communication client interface implementation that uses Generic Communication Framework for communication.

CommServers

The following are the CommServers `.jar` files provided in the `lib` directory:

- `datagram_comServer.jar` - Datagram-based communication server.
- `j2me_httpCommServer.jar` - HTTP-based communication server.
- `serial_commServer.jar` - Serial-based communication server.

Test Export Support Libraries

The following JAR files are automatically copied into the `export` directory when exporting tests and are used from the exported Ant build script, `build.xml`:

- `exportSigner.jar` - Command line tool for signing JAD files when rebuilding exported tests.
- `provisioning_server.jar` - Simple HTTP server used for OTA provisioning of exported tests.

CDC Agents

The following are CDC agent files provided in the `lib` directory:

- `main_agent.jar` - JavaTest agent for Foundation Profile, Personal Basis Profile, and Personal Profile.

  Application model: Main

  Pluggable communication channel
- `xlet_agent.jar` - JavaTest agent for Personal Basis Profile and Personal Profile.

  Application model: Xlet

  Pluggable communication channel
- `applet_agent.jar` - JavaTest agent for Personal Profile.

  Application model: Applet

  Pluggable communication channel

`src` Directory

The `src` directory contains the Java ME TCK Framework test sources, test descriptions, precompiled Java ME TCK Framework classes, and scripts for tests precompilation. Java ME TCK Framework classes consist of server, agent, interview, and communication channel source files.

Java ME TCK Framework Server Classes and Interfaces

The following are Java ME TCK Framework server classes and interfaces provided in the `src` directory:

- `com/sun/cldc/communication/midp/HttpConstants` - HTTP code and string constants.
- `com/sun/cldc/communication/midp/SuiteSigner` - Used to sign JAR files.
- `com/sun/cldc/communication/midp/ContentHandler` - Used to generate the JAD file interface.
- `com/sun/cldc/communication/midp/DefaultContentHandler` - Content handler generating default JAD file.

  A custom content handler can be substituted if device-specific or technology-specific JAD file attributes are needed.
- `com/sun/cldc/communication/midp/BaseServer` - Abstract class.
- `com/sun/cldc/communication/midp/BaseHttpServer` - Used as a basis for HTTP connection tests.
- `com/sun/cldc/communication/midp/HttpServer` - Interface.
- `com/sun/cldc/communication/midp/MIDHttpExecutionServer` - Actual server used for application delivery.

Agent Classes

The following are the agent classes and interfaces provided in the `src` directory:

- `com/sun/cldc/communication/Client` - Communication interface
- `com/sun/cldc/communication/MultiClient` - Communication interface
- `com/sun/tck/midp/javatest/agent/MIDletAgent` - Standard agent
- `com/sun/tck/cldc/javatest/agent/CldcAgent`- Standard agent

Digital Signer

`com/sun/tck/midp/signer/JKSSigner` implements `SuiteSigner` interface. Developers can provide their own version if they are using their own non-standard version of the `SuiteSigner` interface.

Preverification Script

`com/sun/tck/cldc/javatest/PreverificationScript` is used to verify class files when creating a test suite. All class files must be preverified. The preverification script is used during development.

Java ME Technology Version of Harness Classes

The following are Java ME technology versions of the harness classes provided in the `src` directory:

- `com/sun/tck/cldc/lib/Status`
- `com/sun/tck/cldc/lib/MultiTest`
- `com/sun/tck/cldc/lib/Test`

Basic Interview Classes Containing General Questions

The following classes are used as an example of an interview. It is possible to build on these classes if the test suite architect or developer is creating a simple test suite and only adding a few questions. More complex test suites require additional changes not reflected in the following interview classes:

- `com/sun/tck/j2me/interview/distributedtest/DTFInterview`
- `com/sun/tck/midp/interview/ConnectionInterview`
- `com/sun/tck/midp/interview/MidpCldcTckInterview`
- `com/sun/tck/midp/interview/MidpTckBaseInterview`
- `com/sun/tck/midp/interview/MidpTckInterview`
- `com/sun/tck/midp/interview/OTAInterview`
- `com/sun/tck/midp/interview/SigtestInterview`
- `com/sun/tck/midp/interview/TrustedInterview`
- `com/sun/tck/midp/interview/VmAdvancedInterview`
- `com/sun/tck/midp/interview/VmInterview`

Communication Channel

The following classes execute on the server side. These classes are closely related to the communication channel but their purpose is primarily in defining the way in which files are bundled in the JAR file.

- `com/sun/cldc/communication/Server` - Interface.
- `com/sun/cldc/communication/TestProvider` - Generic interface for the test bundler mechanism.
- `com/sun/tck/cldc/javatest/TestBuilder` - Class that packs test classes and resources into `.jar` files.
- `com/sun/tck/cldc/javatest/TestBundler` - Bundler that knows how to package several tests and the agent in a `.jar` file.

  This class keeps track of all resources, conflicts, and `.jar` file size limits.
- `com/sun/tck/cldc/javatest/util/ClassPathReader` - Utility class used to fetch class and resource files from the classpath.
- `com/sun/tck/cldc/javatest/util/JarBuilder` - JAR file builder.

  This is a utility class that creates `.jar` files.
- `com/sun/tck/cldc/communication/TestResultListener` - Implementation of the Communication Channel.
- `com/sun/tck/cldc/communication/GenericTestBundle` - Implementation of the Communication Channel.
- `com/sun/tck/cldc/communication/GenericTestProvider` - Implementation of the Communication Channel.
- `com/sun/tck/cldc/communication/TestBundle` - Implementation of the Communication Channel.
- `com/sun/tck/midp/javatest/MessageClient` - Support for distributed tests on MIDP.

  This class provides an implementation of the harness distribution mechanism on top of HTTP.
- `com/sun/tck/midp/javatest/MidBundler` - MIDP-specific extension that builds on top of the generic CLDC mechanism.
- `com/sun/tck/midp/javatest/MessageSwitch` - Support for distributed tests on MIDP.

  This class provides an implementation of the harness distribution mechanism on top of HTTP.
- `com/sun/tck/midp/javatest/RemoteManager` - Support for distributed tests on MIDP.

  This class provides an implementation of the harness distribution mechanism on top of HTTP.

- `com/sun/tck/midp/javatest/ExportFilter` - Exports tests for offline standalone execution.
- `com/sun/tck/midp/javatest/TestRegistry` - Static class that keeps track of the global properties of individual tests.

  Tests are keyed by their unique IDs.
- `com/sun/tck/midp/javatest/MidBundle` - MIDP-specific extension that builds on top of the generic CLDC mechanism.
- `com/sun/tck/midp/javatest/MidMessageClient` - Support for distributed tests on MIDP.

  This class provides an implementation of the harness distribution mechanism on top of HTTP.

`doc` Directory

The doc directory contains the Java ME TCK Framework Release Notes and the *Java ME TCK Framework Developer's Guide*.

APPENDIX **C**

Test Description Fields and Keywords

This appendix describes the Framework supported fields and keywords for test description files.

The JavaTest harness requires that each test is accompanied by machine readable descriptive data in the form of test suite-specific name-value pairs contained in a test description file. The JavaTest harness uses the contents of the test description file to configure and run tests.

See the Java Technology Test Suite Development Guide for detailed information about creating test description files.

Test Description Fields

The test description file provides the harness with critical information required to run the specified test. Test description files contain fields that supply the following information to the harness:

- Source files that belong to the test
- Class or executable to run
- Information to determine how to run the test

TABLE C-1 lists the test description fields supported by the Framework and describes how their values are used by the harness when the tests are run.

TABLE C-1 Framework Test Description Fields

Field	Description
`title`	A descriptive string that identifies what the test does. The title appears in reports and in the harness status window.
`source`	For compiler tests, contains the names of the files that are compiled during the test run. For runtime tests, contains the names of the files previously compiled to create the test's class files. Precompiled class files are included with the test suite. Source files are included for reference only. Source files are often `.java` files, but can also be `.jasm` or `.jcod` files. • `.jasm` is a low-level bytecode assembler that assembles class files containing sets of bytecodes that are unusual or invalid for use in runtime tests. • `.jcod` is a class-level assembler that builds classes with unusual or invalid structure for use in runtime tests. These tools are used to generate class files that cannot be reliably generated by a Java programming language compiler. For most XML parser tests in the test suite-runtime, the `source` field contains the names of the files that are processed during the test run. These files are XML schema sources and XML documents usually having file name extensions of `.xsd` and `.xml` respectively. Such tests share a single precompiled class, `TestRun`, that invokes the XML parser under test through the Java technology API and passes the source file names to the parser for processing. The test model is similar to compiler testing because the sources used in the tests contain valid and invalid use of various constructs of corresponding languages.
`keywords`	String tokens that can be associated with a given test. They describe attributes or characteristics of the test (for example, how to execute the test, and whether it is a positive or negative test). Keywords are often used to select or deselect tests from a test run. See "Keywords" on page 106.
`executeClass`	The main test class that the harness loads and runs. This class might in turn load other classes when the test is run.

TABLE C-1 Framework Test Description Fields *(Continued)*

Field	Description
`executeArgs`	An array of strings that are passed to the test classes being executed. The arguments might be fixed but often involve symbolic values that are substituted from the test environment (variables defined elsewhere in the test environment). The result of substituting values can be seen in the resulting `.jtr` files. These arguments form the basis for the set of arguments that are passed into the tests defined in the `executeClass` field. The default value of any variable not defined in the test environment is an empty string.
executeLocks	Used in CDC mode.
`timeout`	A value specified in seconds used to override the default ten-minute timeout used with all test suite tests.
`context`	Specifies configuration values required by the test. When a test requires information about the technology under test (context) to determine the expected results, this information is identified in the context field of the test description table. The harness checks to be sure that all values specified in the context field are defined in the test environment before it attempts to execute the test. If any of the values are not set, the test is not executed and the test is considered to be in error. See the *Java Technology Test Suite Development Guide* for detailed information about setting context sensitive properties for a test.
`grant`	Space-separated list of MIDP permission names. Specifies MIDP permissions that must be *granted* for this test application. If tests are run as trusted MIDlets, these permissions are included in the `MIDlet-Permissions` attribute for the test MIDlet. If tests are run as untrusted MIDlets, the tests are filtered out if the security policy does not have all of these permissions granted in the untrusted domain.
`deny`	Space-separated list of MIDP permission names. Specifies MIDP permissions that must be *denied* for this test application. If tests are run as trusted MIDlets, these permissions are not included in the `MIDlet-Permissions` attribute for the test MIDlet. If tests are run as untrusted MIDlets, the test can be filtered out if the security policy has all of these permissions granted in the untrusted domain.

TABLE C-1 Framework Test Description Fields *(Continued)*

Field	Description
`selectIf`	Specifies a condition that must be satisfied for the test to be executed. This field is constructed using environment values, Java programming language literals, and the full set of Boolean operators: (+, -, *, /, <, >, <=, >=, &, |, !, !=, ==). Example: integerValue>=4 & display==”my_computer:0” If the Boolean expression evaluates to false, the test is not run. If the expression evaluates to true, the test is run. If any of the values are not defined in the test environment, the harness considers the test to be in error.
`remote`	Contains information required to run distributed network tests.
`resources`	Contains the location of the resource files that are needed by the test class. The `resources` files can be image, data, or class files. Separate multiple resources with white space.
`remoteSource`	Contains the source name of the remote test class of this distributed test group. Separate multiple remote source files with white space.
`Jad-addon`	Adds a content of a file to the test bundle JAD file.
`Manifest-addon`	Adds a content of a file to the test bundle’s manifest.
`cdcSecurityPermMapper`	
`suitableForPlatform`	
`DoubleDutySecurity`	

Keywords

Keywords are tokens associated with specific tests. Keywords have the following functions:

- Convey information to the harness about how to execute the tests
- Serve as a basis for including and excluding tests during test runs

Users specify keyword expressions in the harness Configuration Editor to filter tests during test runs.

Keywords are specified by the test developer in the keywords field of the test description. Test suites can provide additional custom keywords. TABLE C-2 identifies the Framework keywords and describes the function of their values in the test description file.

TABLE C-2 Framework Keywords

Keyword	Description
`interactive`	Identifies tests that require human interaction.
`negative`	The component under test must terminate with (and detect) an error. An operation performed by a negative test on the component under test must not succeed.
`OTA`	Identifies OTA tests.
`positive`	The component under test must terminate normally. An operation performed by the test on the component under test must succeed.
`runtime`	Identifies tests used with runtime products.
single	Affects the test bundler (tests are bundled one by one).
`trusted`	Identifies tests which must be run in a trusted (operator, manufacturer, and trusted third party) security domain.
`untrusted`	Identifies tests which must be run in an untrusted (unidentified third party) security domain.

Glossary

The definitions in this glossary are intended for Java Compatibility Test Tools (Java CTT) and Java Technology Compatibility Kits (TCK). Some of these terms might have different definitions or connotations in other contexts. This is a generic glossary covering all of Sun's CTTs and TCKs, and therefore, it might contain some terms that are not relevant to the specific product described in this manual.

active agent	A test agent configured to initiate a connection to the JavaTest harness. Active test agents enable you to run tests in parallel using many agents at once and to specify the test machines at the time you run the tests. See also test agent, passive agent, and JavaTest harness agent.
active applet instance	An applet instance that is selected on at least one of the logical channels.
agent monitor	The JavaTest window that is used to synchronize *s* and to monitor agent activity. The Agent Monitor window displays the agent pool and the agents currently in use.
agents	See test agent, active agent, passive agent, and ross.
all values	All of the configuration values required for a test suite. All values include the test environment values specific to that test suite and the JavaTest harness standard values.
API member	Fields, methods and constructors for all public classes that are defined in the specification.
API member tests	Tests (sometimes referred to as class and methods tests) that verify the semantics of API members.
appeals process	A process for challenging the fairness, validity, accuracy, or relevance of one or more TCK tests. Tests that are successfully challenged are either corrected or added to the TCK's exclude list. See also first-level appeals process, second-level appeals process, and exclude list.

Application IDentifier (AID)	An identifier that is unique in the TCK namespace. As defined by ISO 7816-5, it is a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies. There is a unique AID for each package and a unique AID for each applet in the package. The package AID and the default AID for each applet defined in the package are specified in the CAP file. They are supplied to the converter when the CAP file is generated.
Application Management Software (AMS)	Software used to download, store and execute Java applications. Another name for AMS is Java Application Manager (JAM).
Application Programming Interface (API)	An API defines calling conventions by which an application program accesses the operating system and other services.
Application Protocol Data Unit (APDU)	A script that is sent to the test applet as defined by ISO 7816-4.
assertion	A statement contained in a structured Java technology API specification to specify some necessary aspect of the API. Assertions are statements of required behavior, either positive or negative, that are made within the Java specification.
assertion testing	Compatibility testing based on testing assertions in a specification.
automated tests	Test that run without any intervention by a user. Automated tests can be queued and run by the test harness and their results recorded without anyone being present.
behavior-based testing	A set of test development methodologies that are based on the description, behavior, or requirements of the system under test, not the structure of that system. This is commonly known as "black-box" testing.
boundary value analysis	A test case development technique that entails developing additional test cases based on the boundaries defined by previously categorized equivalence classes.

class	The prototype for an object in an object-oriented language. A class might also be considered a set of objects which share a common structure and behavior. The structure of a class is determined by the class variables that represent the state of an object of that class and the behavior is given by a set of methods associated with the class. See also classes.
classes	Classes are related in a class hierarchy. One class might be a specialization (a subclass) of another (one of its superclasses), may be composed of other classes, or might use other classes in a client-server relationship. See also class.
compatibility rules	Define the criteria a Java technology implementation must meet to be certified as "compatible" with the technology specification. See also compatibility testing.
compatibility testing	The process of testing an implementation to make sure it is compatible with the corresponding Java specification. A suite of tests contained in a Technology Compatibility Kit (TCK) is typically used to test that the implementation meets and passes all of the compatibility rules of that specification.
configuration	Information about your computing environment required to execute a Technology Compatibility Kit (TCK) test suite. The JavaTest harness uses a configuration interview to collect and store configuration information.
Configuration Editor	The dialog box used by the JavaTest harness to present the configuration interview.
configuration interview	A series of questions displayed by the JavaTest harness to gather information from the user about the computing environment in which the TCK is being run. This information is used to produce a test environment that the JavaTest harness uses to execute tests.
configuration templates	Files used by the JavaTest harness to configure individual test runs. The JavaTest harness uses the file name extension `*.jti` to store test harness configuration templates.
configuration value	Information about your computing environment required to execute a TCK test or tests. The JavaTest harness uses a configuration interview to collect configuration values.
distributed tests	Tests consisting of multiple components that are running on both the device and the JavaTest harness host. Dividing test components between the device and JavaTest harness is often used for tests of communication APIs, tests that are heavily dependent on external resources, tests designed to run on devices with constrained resources such as a small display, and data transfer tests.
domain	See security domain.

equivalence class partitioning	A test case development technique that entails breaking a large number of test cases into smaller subsets with each subset representing an equivalent category of test cases.
exclude list	A list of TCK tests that a technology implementation is not required to pass in order to certify compatibility. The JavaTest harness uses exclude list files (`*.jtx`), to filter out of a test run those tests that do not have to be passed. The exclude list provides a level playing field for all implementors by ensuring that when a test is determined to be invalid, no implementation is required to pass it. Exclude lists are maintained by the Maintenance Lead and are made available to all technology developers. The ML might add tests to the exclude list for the test suite as needed at any time. An updated exclude list replaces any previous exclude lists for that test suite.
first-level appeals process	The process by which a technology implementor can appeal or challenge a TCK test. First-level appeals are resolved by the Expert Group responsible for the technology specification and TCK. See also appeals process and second-level appeals process.
framework	See test framework.
Graphical User Interface (GUI)	Provides application control through the use of graphic images.
HTML test description	A test description that is embodied in an HTML table in a file separate from the test source file.
implementation	See technology implementation.
instantiation	In object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.
interactive tests	Tests that require some intervention by the user. For example, the user might have to provide some data, perform some operation, or judge whether or not the implementation passed or failed the test.
Java Platform, Standard Edition (Java SE platform)	A set of specifications that defines the desktop runtime environment required for the deployment of Java technology applications. Java SE platform implementations are available for a variety of platforms, but most notably the Solaris and Windows operating systems.

Java Application Manager (JAM)	A native application used to download, store, and execute applications. Another name for JAM is Application Management Software (AMS).
Java Archive (JAR) file	A platform-independent file format that combines many files into one.
Java Compatibility Test Tools (Java CTT)	Tools, documents, templates, and samples that can be used to design and build TCKs. Using the Java CTT simplifies compatibility test development and makes developing and running tests more efficient.
Java Community Process (JCP) program	An open organization of international Java community software developers and licensees whose charter is to develop and revise Java specifications, and their associated Reference Implementation (RI), and Technology Compatibility Kit (TCK).
Java platform libraries	The class libraries that are defined for each particular version of a Java technology in its Java specification.
Java specification	A written specification for some aspect of Java technology.
Java Specification Request (JSR)	The actual descriptions of proposed and final technology specifications for the Java platform.
Java technology	A Java specification and its Reference Implementation (RI). Examples of Java technologies are the J ava SE platform, the Connected Limited Device Configuration (CLDC), and the Mobile Information Device Profile (MIDP).
Java Technology Compatibility Kit	See Technology Compatibility Kit (TCK).
JavaTest harness agent	A test agent supplied with the JavaTest harness to run TCK tests on a Java technology implementation where it is not possible or desirable to run the main JavaTest harness. See also test agent, active agent, and passive agent.
JavaTest harness	A test harness developed by Sun to manage test execution and result reporting for a Technology Compatibility Kit (TCK). The harness configures, sequences, and runs test suites. The JavaTest harness provides flexible and customizable test execution. It includes everything a test architect needs to design and implement tests for implementations of a Java specification.

keywords	Used to direct the JavaTest harness to include or exclude tests from a test run. Keywords are defined for a test by the test suite architect.
Maintenance Lead	The person responsible for maintaining an existing Java specification, related Reference Implementation (RI), and Technology Compatibility Kit (TCK). The ML manages the TCK appeals process, exclude list, and any revisions needed to the specification, TCK, or RI.
methods	Procedures or routines associated with one or more classes, in object-oriented languages.
MultiTest	A JavaTest harness library class that enables tests to include multiple test cases. Each test case can be addressed individually in a test suite exclude list.
namespace	A set of names in which all names are unique.
object-oriented	A category of programming languages and techniques based on the concept of objects, which are data structures encapsulated with a set of routines, called methods that operate on the data.
objects	In object-oriented programming, objects are unique instances of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.
packages	A namespace within the Java programming language. It can have classes and interfaces. A package is the smallest unit within the Java programming language.
passive agent	A test agent configured to wait for a request from the JavaTest harness before running tests. The JavaTest harness initiates connections to passive agents as needed. See also test agent, active agent, and JavaTest harness agent.
prior status	A JavaTest harness filter used to restrict the set of tests in a test run based on the last test result information stored in the test result files (`.jtr`).
Profile specification	A specification that references one of the platform edition specifications and zero or more other Java specifications (that are not already a part of a platform edition specification). APIs from the referenced platform edition must be included according to the referencing rules set out in that platform edition specification. Other referenced specifications must be referenced in their entirety.
Program Management Office (PMO)	The administrative structure that implements the Java Community Process (JCP) program.

protected API	APIs that require that an applet have permission to access them. An attempt to use a protected API without the necessary permissions cause a security exception error.
protection domain	A set of permissions that control which protected APIs an applet can use.
Reference Implementation (RI)	The prototype or proof of concept implementation of a Java specification. All new or revised specifications must include an RI. A specification RI must pass all of the TCK tests for that specification.
second-level appeals process	Allows technology implementors who are not satisfied with a first-level appeal decision to appeal the decision. See also appeals process and first-level appeals process.
security domain	A set of permissions that define what an application is allowed to do in relationship to restricted APIs and secure communications.
security policy	The set of permissions that a technology implementation or Application Programming Interface (API) requires an application to have for the application to access the implementation or API.
signature file	A text representation of the set of public features provided by an API that is part of a finished TCK. It is used as a signature reference during the TCK signature test for comparison to the technology implementation under test.
signature test	Checks that all the necessary API members are present and that there are no extra members that illegally extend the API. It compares the API being tested with a reference API and confirms if the API being tested and the reference API are mutually binary compatible.
specification	See Java specification.
standard values	A configuration value used by the JavaTest harness to determine which tests in the test suite to run and how to run them. The user can change standard values using either the all values or standard values view in the Configuration Editor.
structure-based testing	A set of test development methodologies that are based on the internal structure or logic of the system under test, not the description, behavior, or requirements of that system. This is commonly known as white-box or glass-box testing. Compatibility testing does not make use of structure-based test techniques.
system configuration	Refers to the combination of operating system platform, Java programming language, and JavaTest harness tools and settings.

tag test description	A test description that is embedded in the Java programming language source file of each test.
Technology Compatibility Kit (TCK)	The suite of tests, tools, and documentation that enable an implementor of a Java specification to determine if the implementation is compliant with the specification.
TCK coverage file	A file used by the Java CTT Spec Trac tool to track the test coverage of a test suite during test development. It binds test cases to their related assertion in the specification. The bindings make it possible to generate statistical reports on test coverage.
technology implementation	Any binary representation of the form and function defined by a Java specification.
test agent	An application that receives tests from the test harness, runs them on the implementation being tested, and reports the results to the test harness. Test agents are normally only used when the TCK and implementation being tested are running on different platforms. See also active agent, passive agent, and JavaTest harness agent.
test	The source code and any accompanying information that exercise a particular feature, or part of a feature, of a technology implementation to make sure that the feature complies with the Java specification compatibility rules. A single test can contain multiple test cases. Accompanying information can include test documentation, auxiliary data files, or other resources used by the source code. Tests correspond to assertions of the specification.
test cases	A small test that is run as part of a set of similar tests. Test cases are implemented using the JavaTest harness MultiTest library class. A test case tests a specification assertion, or a particular feature, or part of a feature, of an assertion.
test command	A class that knows how to execute test classes in different environments. Test commands are used by the test script to execute tests.
test command template	A generalized specification of a test command in a test environment. The test command is specified in the test environment using variables so that it can execute any test in the test suite regardless of its arguments.
test description	Machine-readable information that describes a test to the test harness so that it can correctly process and run the related test. The actual form and type of test description depends on the attributes of the test suite. A test description exists

	for every test in the test suite and is read by the test finder. When using the JavaTest harness, the test description is a set of test-suite-specific name-value pairs in either HTML tables or Javadoc tool-style tags.
test environment	One or more test command templates that the test script uses to execute tests and a set of name-value pairs that define test description entries or other values required to run the tests.
test execution model	The steps involved in executing the tests in a test suite. The test execution model is implemented by the test script.
test finder	When using the JavaTest harness, a nominated class, or set of classes, that read, verify, and process the files that contain test descriptions in a test suite. All test descriptions that are located are handed off to the JavaTest harness for further processing.
test framework	Software designed and implemented to customize a test harness for a particular test environment. In many cases, test framework components must be provided by the TCK user. In addition to the test harness, a test framework might (or might not) include items such as a: configuration interview, Java Application Manager (JAM), test agent, test finder, test script, and so forth. A test framework might also include other user-supplied software components (plug-ins) to provide support for implementation-specific protocols.
test harness	The applications and tools that are used for test execution and test suite management. The JavaTest harness is an example of a test harness.
test script	A Java technology software class whose job it is to interpret the test description values, run the tests, and report the results back to the JavaTest harness. The test script must understand how to interpret the test description information returned to it by the test finder.
test specification	A human-readable description, in logical terms, of what a test does and the expected results. Test descriptions are written for test users who need to know in specific detail what a test does. The common practice is to write the test specification in HTML format and store it in the test suite's test directory tree.
test suite	A collection of tests, used with the test harness to verify compliance of the technology implementation to a Java specification. Every Technology Compatibility Kit (TCK) contains one or more test suites.
work directory	A directory associated with a specific test suite and used by the JavaTest harness to store files containing information about the test suite and its tests.

Index