Java™ Technology Compatibility Kit User's Guide Template

For Technology Licensees

Release [VersionNumber]

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A.
1-800-555-9SUN or 1-650-960-1300

May, 2003

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A.
1-800-555-9SUN or 1-650-960-1300

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A.
1-800-555-9SUN or 1-650-960-1300

<NOTE this copyright page states Sun's copyrights regarding this TCK User's Guide Template. It is not intended for use as a template or model for TCK User's Guide copyright statements.>

COPYRIGHT © 2003 SUN MICROSYSTEMS, INC. ALL RIGHTS RESERVED.

SUN MICROSYSTEMS, INC. HEREBY GRANTS A NON-EXCLUSIVE, NON-TRANSFERABLE, WORLDWIDE LICENSE TO JAVA COMMUNITY PROCESS (JCP) MEMBERS TO USE, REPRODUCE, AND CREATE DERIVATIVE WORKS FROM THIS DOCUMENT, SOLELY FOR THE PURPOSE OF CREATING THEIR OWN JAVA TECHNOLOGY COMPATIBILITY KIT USER GUIDES, AND TO DISTRIBUTE, PUBLICLY PERFORM, OR PUBLICLY DISPLAY SUCH USER GUIDES, IN WHOLE OR IN PART, AND IN ANY MEDIA OR FORMAT. LICENSEE AGREES THAT IT MAY NOT MODIFY OR CLAIM ANY LEGAL RIGHTS IN ANY SUN TRADEMARK OR LOGO. LICENSEE MAY NOT USE ANY SUN TRADEMARK OR LOGO EXCEPT IN CONFORMANCE WITH SUN'S TRADEMARK AND LOGO USAGE REQUIREMENTS (WWW.SUN.COM/POLICIES/TRADEMARKS/). THIS LICENSE IS SUBJECT TO AND CONDITIONED UPON LICENSEE'S COMPLIANCE WITH THE TERMS AND CONDITIONS OF THIS LICENSE, AND LICENSEE'S RETENTION, ON ALL REDISTRIBUTIONS, IN WHOLE OR IN PART, OF THE ABOVE COPYRIGHT NOTICE, THIS PERMISSION NOTICE, AND ALL DISCLAIMERS. THE TERMS OF LICENSEE'S USE ARE GOVERNED BY CALIFORNIA LAW, EXCLUDING THAT BODY OF LAW RELATING TO CONFLICTS OF LAWS, AND APPLICABLE FEDERAL LAW, AND MAY ONLY BE AMENDED THROUGH A WRITING SIGNED BY SUN AND LICENSEE.

SUN, SUN MICROSYSTEMS, THE SUN LOGO, JAVA, JAVATEST, JAVA COMMUNITY PROCESS, JCP, J2SE, J2ME, AND JVM ARE TRADEMARKS, REGISTERED TRADEMARKS, OR SERVICE MARKS OF SUN MICROSYSTEMS, INC. IN THE U.S. AND OTHER COUNTRIES. ALL TRADEMARKS ARE USED UNDER LICENSE AND ARE TRADEMARKS REGISTERED IN THE U.S. AND OTHER COUNTRIES.

US GOVERNMENT RIGHTS

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the Sun Microsystems, Inc. licenses and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(ii) (OCT 1988), FAR 12.212(a)(1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as applicable. Sun Microsystems, Inc.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID. IN NO EVENT SHALL SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY DIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, LOSS OF USE, DATA OR PROFITS, OR BUSINESS INTERRUPTION), HOWEVER CAUSED AND UNDER ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, NEGLIGENCE, STRICT LIABILITY, OR TORT, ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. SUN FURTHER DISCLAIMS ANY AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE OR TRADE. NO ADVICE OR INFORMATION, WHETHER ORAL OR WRITTEN, OBTAINED FROM SUN OR ELSEWHERE SHALL CREATE ANY WARRANTY NOT EXPRESSLY STATED IN THIS AGREEMENT. Licensee assumes all risk and liability with respect to use of this document and agrees to indemnify Sun Microsystems, Inc. against any loss, damages, or liability that result from licensee's exercise of these license rights. Sun's rights of indemnification shall survive the termination of this license. Sun may terminate this license at any time if licensee exceeds the scope of the license.

Please

Please

Contents

How to Use This Template

<NOTE: This Preface describes how to use this template. It is not part of the template itself, and should be removed from the book file before creating a document that your customers will see.>

This template is provided for your convenience. Its purpose is to help you create a Technology Compatibility Kit (TCK) Users Guide for your product. Using this template is optional, you are free to write and design your TCK User Guide however you want. You can use all of this template, some of it, or none of it as you wish.

This template assumes that you are basing your TCK on JavaTest™ harness version 3.x. If you are basing your TCK on an earlier version of the JavaTest harness or some other underlying software then some or many portions of this template will not be relevant.

This preface contains the following sections:

User Guide Template Formats

This template is provided in the following formats. Note that some of the features provided in the FrameMaker format may not be available in the other formats.

- **FrameMaker**. All of the FrameMaker files that make up the template, including the FrameMaker book file (`tck.book`) are included in this distribution for your use. The FrameMaker version was created and maintained using Adobe FrameMaker version 6 from Adobe Systems Inc. (`www.adobe.com`).

- **Adobe Acrobat** PDF format (compatible with Acrobat version 4.0 and later). This provides the entire *Technology Compatibility Kit User's Guide Template* in a single PDF file for reference purposes. Text cut and paste operations are supported in this version.
- **ASCII text**. The template is provided in plain ASCII text files that most word processing and desktop publishing applications can load. However, all font and color distinctions are lost in this format. Note the following:
  - In text format each chapter is contained in a separate file. The file `tck_ug_template.txt` lists the files in order with corresponding chapter titles.
  - These text files were created using the FrameMaker "Save As > Text Only" option. ASCII encoding was used (not ANSI).

Template Style Conventions

The following style conventions apply to this template:

Color Codes

The FrameMaker and PDF formats allow text to be displayed in color. (Naturally there are no color codes in the ASCII text version of this template.)

This template uses color as follows:

- Blue. Notes and comments addressed to you the author are displayed in blue (if format allows). They are intended to be removed from the template by you. In FrameMaker format these notes are in conditional text that can be hidden or displayed using the Special > Conditional Text menu item as described in "Working With Conditional Text" on page xviii.
- Red. Optional or example text that you may, or may not, want to include depending on your specific needs are displayed in red (if format allows). Red is also used to describe information that you need to write or include in your manual.

Paragraph Styles

This template uses the following special paragraph styles. The style-name in bold face is the name of the FrameMaker paragraph tag.

<Comment paragraph.> These are notes or comments intended for you the author. They are enclosed in <angle brackets>. If the format allows, these paragraphs are displayed in blue color using courier font. These

paragraphs should all be removed in the final version of your book. Comment paragraphs are also in FrameMaker conditional text so that they can be hidden from view if you wish. This allows you to keep them in the file, but not print out or display them.

***<2b-added** paragraph.> This paragraph style is used to describe the content of information that needs to be written and included by you. For example, specific information describing your product. They are enclosed in <angle brackets>. If the format allows, these paragraphs are displayed blue color and sans-serif italic font. These paragraphs should all be removed in the final version of your book and replaced with the appropriate text.*

Character Styles

This template uses the following special character styles. The style-name in bold face is the name of the FrameMaker character tag.

<WriterNote character tag>. This character style is for explanatory in-paragraph notes or comments addressed to you the author. They are enclosed in <angle brackets> to distinguish them from normal text. If the format allows, they are displayed in blue color and sans-serif font. These comments should all be removed in the final version of your book.

<2b-added character tag>. This character style indicates in-paragraph information that must be added by you. For example, the *<name>* of a particular directory. You must replace what is shown in this style with information appropriate for your book. They are enclosed in <angle brackets> to distinguish them from normal text. If the format permits, this information is displayed in blue color, and italic sans-serif font.

{Optional or Example}. This character style is for optional or example text that you may, or may not, want to include according to your needs. This material is {enclosed in braces} to distinguish it from normal text. If the format allows, they are displayed in magenta color.

If you include the optional or example material, the character format should be reset to the default color. If you do not wish to include this material, it should be removed.

- **Command lines**: Optional command lines are not enclosed in braces because they could be confusing, and that in some cases bullet lines are not enclosed for the same reason.
- **Optional sections**: Where an entire section is optional, only the section title uses the {optional} character format.

Conditional Text

This section applies to the FrameMaker format files only.

The FrameMaker files in this template use FrameMaker conditional text that you can include or exclude from a display, printout, or conversion using the Special > Conditional text menu item. (See "Working With Conditional Text" on page xviii for information on working with conditional text.) The FrameMaker files contain a number of different condition tags:

- **Comment**. Notes and comments addressed to you the author are displayed in "Comment" conditional text. They are not intended for customers, and you need to either delete them or hide them before producing a document that will be seen by customers. Comments are the only conditional text actually used in these template files.
- **Other tags**. The template frame files also contain other conditional tags that have not been used. For example, "Reviewer," "Placeholder," "Deleted," and so on. You can use these as you wish, or create new ones as needed.

Template Variables

This template uses variables for information specific to your product (such as the product or directory names). These variables appear in the template enclosed in [brackets]. You must change the content of these variables to values appropriate for your product. For example, if you are writing a users guide for the ABCP TCK you might change the [NAME TCK] variable to "ABCP TCK".

Depending on the template format you are working with, there are two ways to change the values of these variables:

- **FrameMaker format**. The variables can be edited using the Special > Variable menu item and applied globally to all chapters using the book file as described in your FrameMaker documentation.
- **Other formats**. Globally search and replace the variables on a file by file basis.

The table below lists the special variables used in this template. Note that for FrameMaker users most of the variable names begin with a period so that they sort together in the variables list for convenience in selecting them when editing.

Template Variables

Value	FrameMaker Variable Name	Description
[AgentName]	.agentname	The name of the JavaTest agent used by your product. For example ABCAgent.
[Maintenance Lead]	.company	The name of the technology Maintenance Lead. (Or the name of your company if appropriate.)
[*N.N*]	.techver	The version number of the technology specification. For example 1.1 (Note that this is different than the [VersionNumber] variable which is for the version of the *user guide*, as opposed to the technology. This allows you to have a user guide with a version number different from the technology version number. For example, the technology version might be 1.1 while user guide version number might be 1.1a.
[name]	.lowercase	A lower case product name or acronym for use in file and directory names. For example, if your TCK is for the ABCP specification, you might have filenames like `abcp-tck.jtx`.
[NAME]	.technology	The acronym of the specification the TCK is intended to test. For example ABCP.
[NAME TCK]	.tckname	The name of the TCK. For example ABCP TCK.
[Support Name]	.supportgroup	The name of the expert group supporting the technology.
TCK_DIRECTORY	.TCK_dir	The top-most TCK directory. This is the directory into which the TCK software is installed. Most TCK-related paths are relative to this directory.

Template Variables

Value	FrameMaker Variable Name	Description
[Technology Name]	.techfullname	The full name of the specification (as opposed to an acronym). For example, Absolutely Best Computing Profile. Note that there is a separate variable [*N.N*] for the technology version number. If you want the version number to appear after every instance of the technology name, simply add the version number to the definition of the [Technology Name] variable. (This will require deleting the [*N.N*] variable in the few places it is used in this template.)
[version]	.harnessver	The version number of the test harness.
[Web URL]	.weburl	A web site where users can find additional information, specifications, and support.

In addition to these template-specific variables, Frame users have available a number of standard documentation-oriented variables such as Book Title, Part Number, Version (of the book), and so forth. These can be viewed with the Special > Variable menu item.

Working With Fonts, Variables, and Conditional Text

This section is for those who use FrameMaker.

Working With Conditional Text

If conditional text is displayed on your screen, it will be printed to paper and also included when the document is converted to HTML or PDF format. However, conditional text that has been hidden will not be printed or included when a document is converted. You can hide or display conditional text or change it to regular text as described below.

Hiding Conditional Text

To hide conditional text:

1. **Click Special > Conditional Text in the menu bar.**
2. **Click the Show/Hide button at the bottom right of the Conditional Text dialog box.**
3. **Click the Show radio button.**
4. **Use the arrow buttons to move that conditional tags that you want to hide from the Show box to the Hide box.**
5. **Click the Set button.**
6. **Close the Conditional Text dialog box.**

Displaying Conditional Text

To display conditional text, follow these steps:

1. **Click Special > Conditional Text in the menu bar.**
2. **Click the Show/Hide button.**
3. **Choose which conditional text tags to display:**
   - Choose the Show All radio button to display all conditional text.
   - Chose the Show radio button and use the arrow buttons to select the conditional text tags to be displayed or hidden.
4. **Click the Set button.**
5. **Click the Apply button.**

Converting Conditional Text to Regular Text

To change some text from conditional to regular text in the document follow these steps:

1. **Select (highlight) the Conditional Text that is to be converted to regular text.**
2. **Click Special > Conditional Text in the menu bar.**
3. **Click the Unconditional radio button in the Conditional Text dialog box.**
4. **Click the Apply button.**

Replacing Variables

To replace variables with your technology-specific text, follow these steps:

1. **Click Special > Variable in the menu bar.**

2. **Select (highlight) the variable name in the list of variables.**

   For example, `.technology`.

3. **Click the Edit Definition button.**

4. **Replace the text in the Definition field with the appropriate text.**

   For example, the default definition of the `.technology` variable is [NAME]. You would change that to the name of your technology (without brackets).

5. **Click the Change button to save your change.**

6. **Click the Done button to close the dialog box.**

7. **Close the Conditional Text dialog box.**

Changing Colored Fonts to Black

To change colored text to black:

1. **Select (highlight) the Text that is to changed to regular text.**

2. **If the text is conditional, change it to unconditional.**

   See "Converting Conditional Text to Regular Text" on page xix.

3. **Re-tag the paragraph.**

   If the entire paragraph is in a colored font, you need to re-tag the paragraph. For example, if an entire Bullet1 paragraph is in magenta, you need to open the paragraph catalog and select Bullet1. (This step is not necessary if only a portion of the paragraph is in a colored font.)

4. **Choose Format > Characters > Default Paragraph Font.**

Updating the Document Footer

To update the document footer:

1. **Change the BookTitle variable to the title of your book.**

   "Replacing Variables" on page xix describes how to edit variables. For example, you might change "[Generic Chapter 2 Template]" with "ABCP TCK User Guide."

2. **Change the ReleaseDate variable to the correct release date.**

   For example, you might change "[ReleaseDate]" with "January 2003."

Updating Cross-References

This template contains internal cross references that must be updated after editing for a specific technology. If this document is one chapter of a book, the cross-references are updated each time the book is updated. If this is a stand-alone document, update the internal cross-references as follows:

1. **Click Edit > Update References...in the menu bar.**
2. **Check the All Cross-References box.**
3. **Click the Update button.**

Section 508 Compliance

Section 508 of the federal Rehabilitation Act of 1973 requires that software and documentation be accessible to people with disabilities. Different companies meet the 508 requirements in different ways.

At Sun, the requirements for 508-compliant documentation are met by producing 508-compliant HTML documents. For documents that are written and maintained using FrameMaker, 508-compliant HTML documents are generated from the FrameMaker source files using conversion software tools.

The FrameMaker files for this template include the following features that can be used by a conversion software tool to assist you in producing 508-compliant HTML documents if you wish to do so:

- **First and last list elements.** Special “first” and “last” paragraph tags have been used to identify the first and last paragraphs in bulleted, numbered, step, and table lists. This allows conversion software to identify when a particular list begins and ends. All of these tags are included in the template's catalog.
- **Table and figure description markers.** Where the contents of a figure or table has not been described in the immediately preceding text, special “image” and “table” description markers are inserted in the figure and table captions. Conversion software can be designed to capture the contents of these markers for HTML output. These marker types are included in the FrameMaker file’s list of available markers.

Note – Section 508 requirements are complex. You need to determine how best to handle 508 issues for your product. Use of the paragraph tags and markers supplied with these template FrameMaker files to produce HTML documents will not necessarily meet all 508 requirements for your product documentation. Review the 508 requirements and determine what additional features, if any, are needed. You can review the Guide to the Section 508 Standards for Electronic and Information Technology at `http://www.access-board.gov/sec508/guide/`.

Special Appendices

At the end of this template are two special appendices that are not part of the actual User Guide template.

Appendix D, “Release Notes” provides an optional set of templates that you can use for creating a separate set of Release Notes.

Appendix E, “Parts Bin” contains standard parts such as table and figure templates that you can use as needed.

Indexes

A User Guide of this length should have an index which needs to be created by you using the index tools of your word-processing or desktop-publishing application.

Example Preface

This guide describes how to install, configure, and run the Technology Compatibility Kit (TCK) that is used to test implementations of the *<full_technology_name>* [*N.N*] specification.

The [NAME] TCK is designed as a portable, configurable automated test suite for verifying the compliance of a licensee's implementation of the [NAME] Specification (hereafter referred to as a licensee implementation). The [NAME] TCK uses the JavaTest harness version 3.x to run the test suite.

Note – All references to specific Web URLs are given for the sake of your convenience in locating the resources quickly. These references are always subject to changes that are in many cases beyond the control of the authors of this guide.

Refer to the [Support Name] web site ([Web URL]) for answers to frequently asked questions and send questions you may have to your [Support Name] contact.

Note – The top-most [NAME TCK] installation directory is referred to as *TCK_DIRECTORY* throughout the [NAME TCK] documentation. By default, the *TCK_DIRECTORY* is *base_directory*`/[directory_name]`.

Who Should Use This Book

This guide is for licensees of [Maintenance Lead]'s [NAME] technology to assist them in running the test suite that verifies compliance of their implementation of the [NAME] Specification.

Before You Read This Book

Before reading this guide, you should familiarize yourself with the Java™ programming language and the [NAME] Specification. A good resource for the Java programming language is the Sun Microsystems, Inc. web site, located at: `java.sun.com.`

The [NAME] TCK *<version>* is based on the [NAME] Specification [*N.N*]. Links to the specification and other product information can be found on the Web at: *<web URL>*

[Maintenance Lead] recommends that before documenting your TCK you have read and become familiar with the [NAME] Specification and also the *JavaTest User's Guide* describing the main JavaTest harness. It is located at: *TCK_DIRECTORY*`/doc/javatest/javatest.pdf` in the [NAME] TCK distribution.

`<If your TCK supplies javatest.pdf in a different location, or you supply some other JavaTest documentation, enter the appropriate information above.>`

How This Book Is Organized

If you are installing and using the [NAME] for the first time, [Maintenance Lead] recommends that you read chapters 1 and 2 completely for the necessary background information, and then perform the steps outlined in chapters 3, 4, 5, and 6, while referring to the appendices as necessary.

Chapter 1, "Introduction" gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs) and describes the [NAME TCK]. It also includes a listing of the basic steps needed to get up and running with the [NAME TCK].

Chapter 2, "Procedure for [NAME] [N.N] Certification" describes the conformance testing procedure and testing requirements.

Chapter 3, "Installing the [NAME TCK]" describes [NAME TCK] installation procedures.

Chapter 4, "Starting and Configuring the JavaTest Harness" describes loading the JavaTest harness software and basic [NAME TCK] set up and configuration.

Chapter 5, "Verifying the [NAME TCK]" describes how to start the [NAME TCK] and verify that it is properly configured.

Chapter 6, "Testing Your Implementation" describes how to use the [NAME TCK] to test your implementation.

Chapter 7, "{Testing API Signatures}" describes how to use and run the signature test.

Chapter 8, "{Test-Specific Information}" provides information about the individual tests in the test suite.

Chapter 9, "Debugging Test Problems" describes some methods that can be used to trouble-shoot tests that fail.

Chapter 10, "{Product-Specific Chapter Template}" {provides a template file for any product-specific chapters you require.|

Appendix A, "{Implementing the Test Framework}" {describes the communications channel and how to implement it.}

Appendix B, "Configuration Editor and Environment Variables" {provides one or more tables showing the relationship between JavaTest harness configuration editor questions and the variables they affect.}

Appendix D, "Frequently Asked Questions" provides answers to frequently asked questions.

<There is an additional appendix at the end of this template: Appendix E, "Release Notes" that contains templates for creating release notes. This appendix is not part of an actual TCK User Guide.>

Related Books

- *JavaTest User's Guide* and JavaTest online help (located at *<location>* in the [NAME TCK] distribution)
- The [Technology Name] [*N.N*] specification
- *The Java Programming Language*
- *The Java Language Specification Second Edition*
- *The Java Virtual Machine Specification 2nd Edition, Java 2 Platform*

Accessing Documentation Online

When unzipped and installed, the [NAME TCK] includes a `/doc` directory that contains this manual and the *JavaTest User's Guide*, both in Adobe Acrobat™ PDF format.

<The paragraph above assumes that you use a /doc directory containing both your TCK User Guide and the JavaTest User's Guide.>

The [Support Name] web site ([Web URL]) includes other test-related tools as well. You may also access other Java technology related documentation online at: java.sun.com.

Typographic Conventions Used in This Book

The following table describes the typographic conventions used in this book.

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

CHAPTER 1

Introduction

The [NAME TCK] tests implementations of the [Technology Name] which <describe what it is or does>.

This chapter gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs) and describes the [NAME TCK]. It also includes a listing of what is needed to get up and running with the [NAME TCK]. It contains the following sections:

- <Z_Xref>Compatibility Testing
- <Z_Xref>Java Community Process (JCP) Program and Compatibility Testing
- <Z_Xref>The [NAME TCK]
- <Z_Xref>[NAME TCK]—Configuration
- <Z_Xref>{Underlying Software Implementation}
- <Z_Xref>{Application Management Software (AMS)}
- <Z_Xref>{Other Testing Components}
- <Z_Xref>How the [NAME TCK] Works
- <Z_Xref>{Application Management Software (AMS)}
- <Z_Xref>[NAME TCK]—Getting Started

Compatibility Testing

Java technologies are "*cross-platform*," meaning that they run on different hardware platforms and operating systems. Compatibility testing is the process of testing a technology implementation to make sure that it operates consistently with each platform, operating system, and other implementations of the same Java technology specification.

Therefore, compatibility testing differs from traditional product testing in a number of ways because the focus of compatibility testing is to test those features and areas of an implementation that are likely to differ across other implementations, such as those features that:

- Rely on hardware or operating system-specific behavior.
- Are difficult to port.
- Mask or abstract hardware or operating system behavior.

Compatibility test development for a given feature relies on a complete specification and reference implementation for that feature. Compatibility testing is not primarily concerned with robustness, performance, or ease of use.

Why Compatibility Testing is Important

Java platform compatibility is important to different groups involved with Java technologies for different reasons:

- Compatibility testing is the means by which [Maintenance Lead] ensures that the Java platform does not become fragmented as it is ported to different operating systems and hardware environments.
- Compatibility testing benefits developers working in the Java programming language, allowing them to write applications once and then to deploy them across heterogeneous computing environments without porting.
- Compatibility testing allows application users to obtain applications from disparate sources and deploy them with confidence.
- Compatibility testing benefits Java platform implementors by ensuring a level playing field for all Java platform ports.

TCK Compatibility Rules

Compatibility criteria for all technology implementations are embodied in the TCK Compatibility Rules that apply to a specified technology. Each TCK tests for adherence to these Rules as described in Chapter 2, “Procedure for [NAME] [N.N] Certification”.

TCK Overview

A TCK is a set of tools and tests used to verify that a licensee’s implementation of [Maintenance Lead]’s technology conforms to the applicable specification. All tests in the TCK are based on the written specifications for the Java platform. A TCK tests compatibility of a licensee’s implementation of [Maintenance Lead]’s technology to the applicable specification of the technology. Compatibility testing is a means of ensuring correctness, completeness, and consistency across all implementations developed by [Maintenance Lead] technology licensees.

The set of tests included with each TCK is called the “*test suite.*” Most tests in a TCK’s test suite are self-checking, but some tests require tester interaction. Most tests return either a Pass or Fail status. For a given platform to be certified, all of the required tests must pass. The definition of required tests may change from platform to platform.

The definition of required tests will change over time. Before your final certification test pass, be sure to download the latest Exclude List for the TCK you are using.

Java Community Process (JCP) Program and Compatibility Testing

The Java Community Process℠ (JCP) program is the formalization of the open process that Sun Microsystems, Inc. has been using since 1995 to develop and revise Java technology specifications in cooperation with the international Java community. The JCP℠ program specifies that the following three major components must be included as deliverables in a final Java technology release under the direction of the responsible Expert Group:

- Technology Specification
- Reference Implementation
- Technology Compatibility Kit (TCK)

For further information on the JCP program see this URL: `http://jcp.org`.

The [NAME TCK]

The [NAME TCK] is designed as a portable, configurable, automated test suite for verifying the compliance of a licensee's implementation of [Maintenance Lead]'s [NAME] Specification (JSR-*NNN*). The [Technology Name] specification can be found at: [Web URL].

`<You may want to give a more thorough description or overview of your TCK here.>`

This section describes the [NAME TCK] and covers the following topics:

{[NAME TCK]—New in This Release}

`<This optional section is for new releases of existing TCKs.>`

Among the changes you will find in this version:

1. *<First item in numbered list of new features and changes since the last release>*
2. *<Next item.*
3. *<Last item.*

[NAME TCK] Specifications and Requirements

This section lists the applicable requirements and specifications.

- **[NAME] Version**. The [NAME TCK] *<version>* is based on the [NAME] Specification version [*N.N*].
- **Specification Requirements**. Hardware and software requirements for a [NAME] implementation are described in detail in the [NAME] Specification. Links to the [NAME] specification and other product information can be found at *<web or other location.>*
- **{JavaTest harness**. The [NAME TCK] requires version 3.x of the JavaTest harness.}
- **Reference Implementation**. The designated Reference Implementation for conformance testing of implementations based upon [NAME] Specification [*N.N*] is the [Maintenance Lead] Reference Implementation of the [NAME] specification.
- **Platform requirements**. The following requirements must be met in order to run the [NAME TCK] on the host system:
  - **Operating system**. {You may use any hardware platform that supports Java. [Maintenance Lead] recommends using Solaris, Windows NT, or Windows 98.}
  - **Operating system**. {A J2SE 1.4 compliant platform is required. Sun recommends using J2SE JRE version 1.4 or later on Solaris or Windows 2000. (The [NAME TCK] has been tested with JRE 1.4, other versions such as JRE 1.4.2 have not been tested.)}

`<Two variations of the Operating System bullet are presented above as examples of how you might want to approach this topic.>`

  - **Disk space**. At least *<number>* Megabytes of free disk space available for installation of the [NAME TCK], temporary files, and for the creation of report files.
  - **Memory**. At least *<number>* Megabytes of RAM is recommended for running the [NAME TCK] on Windows platforms.
  - **{Heap memory.** You must insure that your [NAME] implementation provides enough heap memory to run the required tests. (The Sun Reference Implementation proved to work correctly with 292KB of heap memory).}
  - **{Speed**. A minimum speed of 166MHz is required for running the [NAME TCK] on Windows platforms.}

<The bullets below are examples of additional requirements that may (or may not) apply.>

<If additional software is required, list it here. For example, if CLDC and a preverifier are required, you might write:>

- {**Connected Limited Device Configuration**. CLDC version 1.0 for Windows 2000 is required for running the designated Reference Runtime.}
- {**Preverifier**. The designated Reference Preverifier for the conformance testing of implementations based upon [NAME] Specification [*N.N*] is the preverifier included with the Reference Runtime.}
- {**Java Communications API 2.0 implementation**. If the device under test supports communication via serial ports using `javax.microedition.io.CommConnection` functionality, a Java Communications API 2.0 implementation capable of communicating with the device must be available.}
- {**Network Control Interfaces (NCI) implementation**. The WMA-related portion of the TCK implements distributed tests with assistance from a component called the Network Control Interfaces (NCI) implementation.}

[NAME TCK] Components

This section describes the main components that make up the [NAME TCK].

JavaTest Harness

The JavaTest™ harness version 3.x is a set of tools designed to run and manage test suites on different Java platforms. The JavaTest harness can be described as both a Java application and a set of compatibility testing tools. It can run tests on different kinds of Java platforms and it allows the results to be browsed on-line within the JavaTest GUI, or off-line in the HTML reports that the JavaTest harness generates.

The JavaTest harness includes the applications and tools that are used for test execution and test suite management. It supports the following features:

- Sequencing of tests, allowing them to be loaded and executed automatically.
- Graphic user interface (GUI) for ease of use.
- Automated reporting capability to minimize manual errors.
- Failure analysis
- Test result auditing and auditable test specification framework.
- Distributed testing environment support.

To run tests using the JavaTest harness, you specify which tests in the test suite to run, how to run them, and where to put the results as described in <Z_Xref>Chapter 4, “Starting and Configuring the JavaTest Harness.

<Choose one of the two example paragraphs below as appropriate for your TCK. The difference is that the second one allows for running TCK tests against an implementation running on an emulated environment or breadboard. The conformance rules may, or may not, allow this.>

<For some TCKs the compatibility rules allow running the TCK against an emulation or breadboard equivalent. In other cases, the test must be run against an actual device. The two sample paragraphs below address these two cases.>

{The JavaTest harness runs tests on a [NAME] target device or the [NAME] Reference Implementation running on a *platform* system.}

{The JavaTest harness runs tests on your target device or on an equivalent emulated environment or a breadboard, or the [NAME] Reference Implementation running on a *platform* system.}

TCK Compatibility Rules

Compatibility criteria for a technology implementation are identified in the TCK Compatibility Rules. The TCK tests a technology implementation to make sure that it adheres to those rules.

The compatibility rules that apply to [NAME] are described in <Z_Xref>Chapter 2, “Procedure for [NAME] [N.N] Certification.”

TCK Compatibility Test Suite

The *test suite* is the collection of tests used by the JavaTest harness to test a particular technology implementation. In this case, it is the collection of tests used by the [NAME TCK] to test an implementation of the [NAME] implementation. The tests are designed to verify that a licensee’s run-time implementation of the technology complies with the appropriate specification. The individual tests correspond to assertions of the specification.

The tests that make up the TCK compatibility test suite are precompiled and indexed within the TCK test directory structure. When a test run is started, the JavaTest harness scans through the set of tests that are located under the directories that have been selected. While scanning, the JavaTest harness selects the appropriate tests according to any matches with the filters you are using and queues them up for execution.

Signature Testing

The Static signature test verifies signatures of APIs which are in a `.jar` archive format or the directory hierarchy. These APIs are regarded as identical or at least equivalent to APIs which are available in the [NAME] implementation under test. (See <Z_Xref>“Signature Test” on page 9 for a more detailed discussion of signature tests.)

<The common practice is to now include the signature test as part of the rest of the test suite. However, in some cases a TCK might be designed in a way that requires running the Signature Test outside of the test suite. For example, it might have to be run from the command line. If this is the case, delete this section.>

{Test Agents}

<In many cases, the technology being tested and the JavaTest harness are both running on the same system. In which case, no test agent is needed and this section is not necessary.>

<But some TCKs allow (or require) the JavaTest harness and the technology being tested to be run on different systems. In such cases you use test agents. For some TCKs you can use the JavaTest Agent provided with the JavaTest harness, or you can create your own custom agents. This optional section is used to describe your test agent.>

If the JavaTest harness and the technology you are testing are being run on different systems, you must install an agent on the system being tested.

A test agent is a small Java™ program that is used in conjunction with the JavaTest harness to run tests on a Java™ platform on which it is not possible or desirable to run the main JavaTest harness. The agent running on the test platform responds to requests from the JavaTest harness running elsewhere.

<The optional paragraph below is for TCKs using the standard JavaTest Agent.>

The [NAME TCK] uses the JavaTest Agent provided with the JavaTest harness.

<The optional paragraph below is for TCKs using a custom agent.>

The [NAME TCK] uses a custom agent called [AgentName].

To perform [NAME TCK] tests on a platform that is not running the JavaTest harness, you:

- Set up and run the JavaTest harness on one system.
- Connect the system running the JavaTest harness to the system or device being tested.
- Install and run the agent on the device being tested.

Under this arrangement, the JavaTest harness running on a PC or workstation performs the following functions:

1. Specifies the tests that are to be run.
2. Issues commands to the [AgentName] to run the tests on the device being tested.
3. Writes the results of those tests to the various report files on the PC or workstation where the JavaTest harness resides.

Exclude Lists

Each version of a TCK includes an Exclude List contained in a `.jtx` file. This is a list of test file URLs that identify tests which do not have to be run for the specific version of the TCK being used. Whenever tests are run, the JavaTest harness automatically excludes any test on the Exclude List from being executed.

A licensee is not required to pass any test—or even run any test—on the Exclude List.

The Exclude List file included in the [NAME TCK] is located in the `lib` directory and has a file name based on the corresponding [NAME TCK] version number. For example, the Exclude List for version *<1.0>* of the [NAME TCK] is *TCK_DIRECTORY*`/lib/[name]-tck_`*<10>*`.jtx`.

Note – From time to time, updates to the Exclude List are made available on the *<location>* web site. You should always make sure you are using an up-to-date copy of the Exclude List before running the [NAME TCK] to verify your implementation.

A test might be included in an Exclude List for reasons such as:

- An error in an underlying implementation API has been discovered which does not allow the test to execute properly.
- An error in the specification that was used as the basis of the test has been discovered.
- An error in the test itself has been discovered.
- The test fails due to a bug in the tools (such as the JavaTest harness, for example).

In addition, all tests are also tested against the [Maintenance Lead] Reference Implementation (RI). Any tests that fail when run on a reference Java platform are put on the Exclude List. Any test that is not specification-based, or for which the specification is vague, may be excluded. Any test that is found to be implementation dependent (based on a particular thread scheduling model, based on a particular file system behavior, and so on) may be excluded.

Note – Licensees are not permitted to alter or modify Exclude Lists. Changes to an Exclude List can only be made by using the procedure described in <Z_Xref>“[NAME] Test Appeals Process” on page 21

Result Files

<The example paragraph below assumes you are using JavaTest harness 3.x or later. If you are using something else, you need to describe whatever result files (if any) are produced.>

The information that is presented in the JavaTest harness Test Summary Browser is also written to files called JavaTest result files. These result files can be found in the work directory that you specified as part of your test run. The JavaTest harness creates a test result hierarchy similar to the test suite hierarchy that contains your tests. Result files for the tests in your test run appear under the appropriate directory within this result hierarchy.

See <Z_Xref>“Exclude File Format” on page 88 for more information about these files.

Signature Test

<In most TCKs the signature test is automatically included in a normal test run. In other TCKs the signature test has to be run separately from the command line. This example section is for cases where the signature test is run automatically along with all the other tests. <Z_Xref>Chapter 7, “{Testing API Signatures},’ is for cases where the signature test is run separately. The assumption is that you would use either this section, or Chapter 7, as appropriate, but not both. If the Signature Test is not included as part of the test suite and has to be run separately, edit the paragraphs below as needed and make a cross-reference to <Z_Xref>Chapter 7, “{Testing API Signatures}.>

It is physically impossible to verify API signatures using the [Technology Name] implementation on the target device in the absence of reflection capabilities. However, API libraries which are burned into ROM or placed into device in some other way, typically have prototypes that follow the standard class file format. In this case it is possible to explore the class files of the prototype to gain confidence that the API libraries that are actually present on the target device comply with the specification.

The [NAME TCK] contains a Static signature test exactly for this purpose. This test is executed by the JavaTest harness as part of a standard test run. (It is also possible to run this test separately from the command line.)

The Static signature test verifies signatures of APIs which are in a `.jar` archive format or the directory hierarchy. These APIs are regarded as identical or at least equivalent to APIs which are available in the [NAME] implementation under test.

{Components Provided by User}

<User-provided components will vary. Use this optional section to describe the user-provided components required by the [NAME TCK]. The bullets below are examples taken from various TCKs that may, or may not, be relevant to your TCK.>

{You must provide the following TCK components:}

- {**Communications channel**. As defined by the MIDP 2.0 specification, your implementation must provide Application Management Software (AMS) as described in *<cross-reference to appendix>*.}
- {**Network Control Interfaces**. The WMA-related portion of the TCK implements distributed tests with assistance from a component called the Network Control Interfaces (NCI) implementation. The NCI has to be implemented by you as described in *<cross-reference to appendix>*."}
- {**Preverifier script**. If you run your preverifier on a remote host you need to write a script that handles the preverifier files as described in *<cross-reference to section>*.}

[NAME TCK]—Configuration

You use the JavaTest harness Graphic User Interface (GUI) to configure test runs. as described in <Z_Xref>Chapter 4, “Starting and Configuring the JavaTest Harness."

{Underlying Software Implementation}

<Some specifications require that some underlying software implementation also be present on the device. If that is the case, then include this optional section.>

An implementation of the [NAME] [VersionNumber] specification requires some underlying software that conforms to other Java specifications as follows:

<The bullets below are examples that may, or may not, apply to your TCK. Use or adapt them as appropriate.>

- **MIDP plus CLDC**. Conformance to the Mobile Independent Device Profile (MIDP) 1.0 or 2.0 plus CLDC 1.0 or 1.1 specifications as appropriate. In this case, the [NAME TCK] assumes that your MMAPI implementation includes valid implementations of both the MIDP and CLDC specifications that have been tested by the appropriate MIDP TCK.
- **CLDC plus IllegalStateException**. Conformance to the Connected Limited Device Configuration (CLDC) 1.0 or 1.1 specifications plus a profile that includes the IllegalStateException. In this case, the [NAME TCK] assumes that your MMAPI implementation includes a valid implementations of the CLDC specification plus some kind of profile the provides the IllegalStateException as tested by the appropriate CLDC TCK. (In this case you may need to provide some additional software as described in <Z_Xref>Appendix A, "{Implementing the Test Framework}.)

The [NAME] Reference Implementation (RI) supplied by Sun Microsystems uses *<whatever>* as its underlying technology. This *[NAME] Technology Compatibility Kit User's Guide* describes how to configure and run the [NAME TCK] against the [NAME] RI as an example, or model, of how you would set it up and use it to test your implementation.

{Application Management Software (AMS)}

<Some TCKs that run tests on small devices make use of Application Management Software (AMS). This optional section is for these cases.>

Some application management code is required to be present on the target device in order for it to receive the [AgentName] and test bundles from the JavaTest harness. This is called Application Management Software (AMS). In some contexts AMS is referred to as Java Application Manager (JAM).

The AMS component must be supplied by you for your implementation.

The AMS component is typically written in native code. Due to significant variations and feature differences among potential [NAME] devices, the details of application management are highly device-specific and implementation-dependent.

See <Z_Xref>Appendix A, "{Implementing the Test Framework}," for information about what you need to supply for [NAME] implementations.

{Other Testing Components}

<This optional section is for describing any additional software components required by your TCK but not supplied as part of the TCK.>

How the [NAME TCK] Works

<This section is for providing an narrative overview of how your TCK works. The example provided below is taken from a TCK running on a workstation and conducting tests on a J2ME device using a custom agent. Your description may be quite different.>

When using the [NAME TCK] to test a [NAME] implementation, the [NAME] implementation is normally run on a target device that is connected in some manner to a PC or a workstation running the JavaTest harness. Under this arrangement, the [NAME] is said to run in a "*JavaTest-Agent*" set up.

The JavaTest harness is run on the PC or workstation using a certified Java 2 Standard Edition Runtime. Its responsibility is to package groups of tests into JAR files (test bundles) and make them available to the target device running the implementation being tested.

The JavaTest harness performs the following functions:

- Downloads the ([AgentName]) to the target device.
- Sends test bundles to the target device.
- Converts them to the application format of the target device, if necessary.}
- Dispatches tests for execution to the target device.
- Receives test results back from the target device.

The agent [AgentName] is pre-packaged with each test bundle. It runs on the target device. Its task is to execute tests from this bundle on the target device and to report results back to the JavaTest harness. The agent acts as a *client* to communicate with JavaTest harness.

The server and the client are Java technology implementations of the interfaces defined in the [NAME TCK].

Test results are displayed by the JavaTest harness.

About [NAME TCK] Tests

<Use this section to provide a brief overview of the [NAME TCK] tests. For example, your test suite might include both interactive tests that require some user intervention and automated tests that require no user intervention.>

<The sample text below is adapted from the MIDP case, and is provided here as an example of one way of approaching this section. This example section assumes that you are using the JavaTest harness, if you are using some other harness you will have to modify the descriptions as appropriate.>

This section provides general information about running [NAME TCK] tests. It contains the following sub-sections:

- <Z_Xref>“{Types of Tests}” on page 14
- <Z_Xref>“{Interactive Tests}” on page 14
- <Z_Xref>“Signature Testing” on page 7

[NAME TCK] tests are designed to verify that a licensee’s runtime implementation of the technology complies with the appropriate specifications. The individual tests correspond to assertions of the specification.

A test is a collection of related classes and resources. Each test executes one or more *test cases*. A test case exercises a component of the implementation in order to verify that the component’s behavior is complaint with the spec. Each test case returns a status indicating success or failure. The tests are combined into a test suite. In other words, the [NAME] test suite is simply the collection of [NAME TCK] tests.

The tests are intended to be independent of any particular licensee’s implementation of the [NAME] technology specifications. Most tests are self-checking, self-contained, and self-sufficient. This means that the tests provide their own pass/fail result, are not dependent on other tests (all of the code resides in the same file), and do not rely on external classes except for the [NAME] and [NAME TCK] framework classes.

The tests use standard status returns. The [NAME TCK] tests are launched from JavaTest harness.

The tests that make up the TCK compatibility test suite are precompiled and indexed within the TCK test directory structure to form the test suite. Before a test run is started, the JavaTest harness scans through the set of tests that are located under the directory that has been selected. While scanning, the JavaTest harness selects the appropriate tests according to any matches with the filters you are using and queues them up for execution.

Note – Multiple test runs with different test selection criteria may be required to completely and correctly test an implementation.

{Types of Tests}

<Some TCKs include both automatic and interactive tests, others do not. Use this optional section to describe the kind of tests in your TCK.>

[NAME TCK] tests are either automated or interactive:

- **Automated tests**: Do not require user interaction to run. Automated tests can be specified by choosing the `Automated` radio button in the JavaTest harness 3.x Configuration Editor's Test Subset question.
- **Interactive tests**: Require some user input or other kind of interaction. The [NAME TCK] contains two kinds of interactive test that use different methods to report test results:
  - Tests that require users to view the results on the device screen and judge a pass / fail result.
  - Tests where the test itself reports a pass / fail result, but still require user interaction.

  Interactive tests can be specified by choosing the `Interactive` radio button in the JavaTest harness 3.x Configuration Editor's Test Subset question.

  See for detailed information on how to select which tests to run.

{Interactive Tests}

Interactive tests are tests of a graphic user interface (GUI), or other elements that require user-participation or user-judgement of pass / fail results. These tests utilize the distributed framework to separate the test instructions and controls from the test running on the device (or equivalent).

See <Z_Xref>"{[Optional] Running Interactive Tests}" on page 51 for more information.

[NAME TCK]—Getting Started

This section provides an general overview of what needs to be done to install, set up, test, and use the [NAME TCK]:

1. **Make sure that a certified Java 2 Standard Edition Runtime environment has been correctly installed on the system you want to host the JavaTest harness.**

   This should be the version described in the Release Notes. Consult the corresponding J2SE Runtime environment documentation for installation instructions.

2. **Install the [NAME TCK] on the system that you want to use as the test harness host system.**

   This system has to be able to access the device through the HTTP protocol.

<If you are using some other protocol or method of establishing communication between the test harness and the device, edit the paragraph above as appropriate.>

   This system must have J2SE *<version>* or higher installed.

<In most cases installing the TCK as described in the next step also installs the test harness. If so, use the bullet below. If not, omit the bullet below and describe what needs to be done to install the test harness.>

   Installation of the [NAME TCK] includes installation of the JavaTest harness. See <Z_Xref>Chapter 3, “Installing the [NAME TCK],” for more information.

3. **{List installation of any other software as appropriate.>**

   {Consult the documentation for each of these software application for installation instructions.}

4. **Install the implementation of [NAME] [*N.N*] that is under test is on the appropriate host platform that the test harness can access through the appropriate protocol.**

   - **RI:** Consult the [NAME] Reference Implementation documentation for installation instructions.
   - **Device:** Consult your implementation documentation for installation instructions.

5. **Start up and configure the JavaTest harness.**

   Use the JavaTest harness configuration editor interview to enter the basic configuration parameters it needs. (See <Z_Xref>Chapter 4, “Starting and Configuring the JavaTest Harness.")

6. **Verify the JavaTest harness Installation.**

   Test that the JavaTest harness is running correctly with a simple series of tests. (See <Z_Xref>“Verifying Installation and Setup” on page 39.)

<Some technology implementations require additional software to be loaded on the device being tested. For example, a TCK might require the presence of Connected Limited Device Configuration (CLDC). The next step instructs the user to load and test such software.>

7. **{Install, set up, and test** *<any additional required software>}*

   {(See <Z_Xref>“{Verifying Other Required Components}” on page 40.)}

<The example shown below is for CLDC.>

This step requires that your CLDC implementation on the [NAME] device pass the CLDC TCK compatibility tests before proceeding to the [NAME TCK] tests.

Note that if you make any changes to the CLDC implementation after running the CLDC TCK, the CLDC TCK tests have to be run again.

8. **Test the Full [NAME] Implementation on the target device**

   Test the full [NAME] implementation installation by running the entire test suite on the target device. (See <Z_Xref>“Using the [NAME TCK] to Test a Product” on page 49.)

CHAPTER 2

Procedure for [NAME] [*N.N*] Certification

<JCP Expert Groups can use this template in the preparation of Chapter 2 for TCK User's Guides. The contents of this chapter are provided as examples only and can be modified as required to support your technology.>

This chapter describes the compatibility testing procedure and compatibility requirements.

Certification Overview

- Install the appropriate version of the Technology Compatibility Kit (TCK) and execute it in accordance with the instructions in this User's Guide.
- Ensure that you meet the requirements outlined in "Compatibility Requirements," below.

Compatibility Requirements

This section described the compatibility rules and defines the terms used in those rules.

<When modifying, adding, or removing rules in "Rules for [NAME] Products" on page 20," review the following definitions for required additions, deletions, or modifications.>

Definitions

These definitions are for use only with these compatibility requirements and are not intended for any other purpose.

TABLE 1 Definitions

Term	Definition
Computational Resource	A piece of hardware or software that may vary in quantity, existence, or version, which may be required to exist in a minimum quantity and/or at a specific or minimum revision level so as to satisfy the requirements of the Test Suite. Examples of computational resources that may vary in quantity are RAM and file descriptors. Examples of computational resources that may vary in existence (that is, may or may not exist) are graphics cards and device drivers. Examples of computational resources that may vary in version are operating systems and device drivers.
Conformance Tests	All tests in the Test Suite for an indicated Technology Under Test, as distributed by the Maintenance Lead, excluding those tests on the Exclude List for the Technology Under Test.
Documented	Made technically accessible and made known to users, typically by means such as marketing materials, product documentation, usage messages, or developer support programs.
Exclude List	The most current list of tests, distributed by the Maintenance Lead, that are not required to be passed to certify conformance. The Maintenance Lead may add to the Exclude List for that Test Suite as needed at any time, in which case the updated Exclude List supplants any previous Exclude Lists for that Test Suite.
Libraries	The class libraries, as specified through the Java Community Process^{SM} (JCP^{SM}), for the Technology Under Test. The Libraries for [NAME] are listed at the end of this chapter.
Location Resource	A location of classes or native libraries that are components of the test tools or tests, such that these classes or libraries may be required to exist in a certain location in order to satisfy the requirements of the test suite. For example, classes may be required to exist in directories named in a CLASSPATH variable, or native libraries may be required to exist in directories named in a PATH variable.
Maintenance Lead	The JCP member responsible for maintaining the Specification, reference implementation, and TCK for the Technology. [Maintenance Lead] is the Maintenance Lead for [NAME].

TABLE 1 Definitions

Term	Definition
Operating Mode	Any Documented option of a Product that can be changed by a user in order to modify the behavior of the Product. <Modify the following example for your technology:> {For example, an Operating Mode of a Runtime can be binary (enable / disable optimization), an enumeration (select from a list of localizations), or a range (set the initial Runtime heap size).}
Product	A licensee product in which a Runtime is implemented or incorporated, and that is subject to compatibility testing.
Product Configuration	A specific setting or instantiation of an Operating Mode. <Modify the following example for your technology:> {For example, a Runtime supporting an Operating Mode that permits selection of an initial heap size might have a Product Configuration that sets the initial heap size to 1 Mb.}
Resource	A Computational Resource, a Location Resource, or a Security Resource.
Rules	These definitions and rules in this Compatibility Requirements section of this User's Guide.
Security Resource	A security privilege or policy necessary for the proper execution of the Test Suite. For example, the user executing the Test Suite will need the privilege to access the files and network resources necessary for use of the Product.
Specifications	The documents produced through the JCP that define a particular Version of a Technology. The Specifications for the Technology Under Test can be found later in this chapter.
Technology	Specifications and a reference implementation produced through the JCP.
Technology Under Test	<If appropriate, add the number after the [NAME].> Specifications and the reference implementation for [NAME].
Test Suite	The requirements, tests, and testing tools distributed by the Maintenance Lead as applicable to a given Version of the Technology.
Version	A release of the Technology, as produced through the JCP.

Rules for [NAME] Products

For each version of an operating system, software component, and hardware platform Documented as supporting the Product:

<The following rules are numbered using the character "T" to designate that they are Template rules. In technology specific documents, the applicable rules should be sequentially numbered for that document and preceded by one or more unique character designator(s) for that technology; such as MIDP for MIDP rules. Use the Paragraph Designer to change the character designator(s) in the numbering properties field for the appropriate paragraph tag.>

<Note to Maintenance Leads: All exception rules should require documentation of any operating mode invoking the exception.>

<Note to Maintenance Leads: All exception rules should require that at least one mode pass all the tests.>

T1. {The Product must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests, in every Product Configuration and in every combination of Product Configurations, except only as specifically exempted by these Rules.}

{For example, if a Product provides distinct Operating Modes to optimize performance, then that Product must satisfy all applicable compatibility requirements for a Product in each Product Configuration, and combination of Product Configurations, of those Operating Modes.}

T1.1 {If an Operating Mode controls a Resource necessary for the basic execution of the Test Suite, testing may always use a Product Configuration of that Operating Mode providing that Resource, even if other Product Configurations do not provide that Resource. Notwithstanding such exceptions, each Product must have at least one set of Product Configurations of such Operating Modes that is able to pass all the Conformance Tests.}

{For example, a Product with an Operating Mode that controls a security policy (i.e., Security Resource) which has one or more Product Configurations that cause Conformance Tests to fail may be tested using a Product Configuration that allows all Conformance Tests to pass.}

T1.2 {A Product Configuration of an Operating Mode that causes the Product to report only version, usage, or diagnostic information is exempted from these compatibility rules.}

T1.3 {A Product may contain an Operating Mode that selects the Edition with which it is compatible. The Product must meet the compatibility requirements for the corresponding Edition for all Product Configurations of this Operating Mode. This Operating Mode must affect no smaller unit of execution than an entire Application.}

T2. {Some Conformance Tests may have properties that may be changed. Apart from changing such properties no source or binary code for a Conformance Test may be altered in any way without prior written permission. Any such allowed alterations to the Conformance Tests would be posted to the [Support Name]web site and apply to all licensees.}

T3. {The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.}

T4. {The Exclude List associated with the Test Suite cannot be modified.}

T5. {The Maintenance Lead can define exceptions to these Rules. Such exceptions would be made available to and apply to all licensees.}

T6. {All hardware and software component additions, deletions, and modifications to a Documented supporting hardware/software platform, that are not part of the Product but required for the Product to satisfy the compatibility requirements, must be Documented and available to users of the Product.}

{For example, if a patch to a particular version of a supporting operating system is required for the Product to pass the Conformance Tests, that patch must be Documented and available to users of the Product.}

T7. {The Product must contain the full set of public and protected classes and interfaces for all the Libraries. Those classes and interfaces must contain exactly the set of public and protected methods, constructors, and fields defined in the Specifications for those Libraries. No subsetting, supersetting, or modifications of the public and protected API of the Libraries are allowed except only as specifically exempted by these Rules.}

T7.1 {If a Product includes Technologies in addition to the Technology Under Test, then it must contain the full set of combined public and protected classes and interfaces. The API of the Product must contain the union of the included Technologies. No further subsetting, supersetting, or modifications to the APIs of the included Technologies are allowed.}

T8. {Except for tests specifically required by this TCK to be recompiled (if any), the binary Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.}

T9. {The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.}

[NAME] Test Appeals Process

<This is an example of an appeals process which can be modified as necessary.>

The Maintenance Lead will be the point of contact for all test challenges to the Test Suite for the [NAME].

If a test is determined to be invalid in function or if its basis in the specification is suspect, the test may be challenged by any licensee of the [NAME TCK]. Each test validity issue must be covered by a separate test challenge. Test validity or invalidity will be determined based on its technical correctness such as:

1. Test has bugs (i.e., program logic errors)
2. Specification item covered by the test is ambiguous
3. Test does not match the specification
4. Test assumes unreasonable hardware and/or software requirements
5. Test is biased to a particular implementation

Challenges based upon issues unrelated to technical correctness as defined by the specification will normally be rejected.

Test challenges must be made in writing to [Support Name] and include all relevant information as described in the Test Challenge form below. The process used to determine the validity or invalidity of a test (or related group of tests) is described in "[NAME] TCK Test Appeals Steps" on page 22."

All tests found to be invalid will either be placed on the Exclude List for that version of the [NAME] TCK or have an alternate test made available as follows:

- Tests that are placed on the Exclude List will be placed on the Exclude List within one business day after the determination of test validity. The new Exclude List will be made available to all [NAME] TCK licensees on the [NAME] TCK web site.
- The Maintenance Lead has the option of creating alternative tests to address any challenge. Alternative tests (and criteria for their use) will be made available on the [NAME] TCK web site.

Note – Passing an alternative test is deemed equivalent to passing the original test.

▼ [NAME] TCK Test Appeals Steps

1. **[NAME] licensee writes a test challenge to the Maintenance Lead contesting the validity of one or a related set of [NAME] tests.**

   A detailed justification for why each test should be invalidated must be included with the challenge as described by the Test Challenge form below.

2. **The Maintenance Lead evaluates the challenge.**

   If the appeal is incomplete or unclear, it is returned to the submitting licensee for correction. If all is in order, the Maintenance Lead will check with the test developers to review the purpose and validity of the test before writing a response. The Maintenance Lead will attempt to complete the response within 5 business

days. If the challenge is similar to a previously rejected test challenge (i.e., same test and justification), the Maintenance Lead will send the previous response to the licensee.

3. **The challenge and any supporting materials from test developers is sent to the specification engineers for evaluation.**

   A decision of test validity or invalidity is normally made within 15 working days of receipt of the challenge. All decisions will be documented with an explanation of why test validity was maintained or rejected.

4. **The licensee is informed of the decision and proceeds accordingly.**

   If the test challenge is approved and one or more tests are invalidated, the Maintenance Lead places the tests on the Exclude List for that version of the [NAME] (effectively removing the test(s) from the Test Suite). All tests placed on the Exclude List will have a bug report written to document the decision and made available to all licensees through the bug reporting database on the [Support Name] web site. If the test is valid but difficult to pass due to hardware or operating system limitations, the Maintenance Lead may choose to provide an alternate test to use in place of the original test (all alternate tests are made available to the licensee community).

5. **If the test challenge is rejected, the licensee may choose to escalate the decision to the Executive Committee (EC), however, it is expected that the licensee would continue to work with the Maintenance Lead to resolve the issue and only involve the EC as a last resort.**

TABLE 2 Test Challenge Form

Test Challenger Name and Company
Specification Name(s) and Version(s)
Test Suite Name and Version
Exclude List Version
Test Name
Complaint (argument for why test is invalid)

TABLE 3 Test Challenge Response Form

Test Defender Name and Company
Test Defender Role in Defense (e.g., test developer, Maintenance Lead, etc.)
Specification Name(s) and Version(s)
Test Suite Name and Version
Test Name
Defense (argument for why test is valid)
-can be iterative-
Implications of test invalidity (e.g., other affected tests and test framework code, creation or exposure of ambiguities in spec (due to unspecified requirements), invalidation of the reference implementation, creation of serious holes in test suite)
Alternatives (e.g., are alternate test appropriate?)

Specifications for [NAME]

The Specifications for [NAME] are found on the JCP web site.

Libraries for [NAME]

<The following is a format example only and must be replaced with the complete, current Libraries listed for your Technology:>

- java.applet
- java.awt
- java.awt.color
- java.awt.datatransfer
- java.awt.dnd
- java.awt.dnd.peer
- java.awt.event

CHAPTER 3

Installing the [NAME TCK]

This chapter describes [NAME TCK] installation procedures. It contains the following sections:

Obtaining the Software

<Use this section to describe how and where to obtain the TCK software. At a minimum, you should answer the following questions:
1. Where and how the software is obtained. CD? Web?
2. Where are installation instructions are obtained?>

Installing the Software

<Use this section to describe the TCK installation procedure. For example, unzipping an archive, un-tarring a tar file, running an executable script or program, or whatever. It should include step by step instructions.>

[NAME TCK] Contents

<The example text below assumes a typical post-installation directory structure.>

The top most [NAME TCK] installation directory, is referred to as *TCK_DIRECTORY* throughout the [NAME TCK] documentation. You can name this directory whatever you want.

Once the [NAME TCK] is installed, several directories will be created under the *TCK_DIRECTORY/*. The contents of these directories are as follows (on Win32 platforms assume backslashes in directory paths, instead of forward slashes used here). The JavaTest documentation is also available in PDF format, for easy online viewing or for printing. This file is named `javatest.pdf` and it resides in the *TCK_DIRECTORY*`/doc/javatest/` directory

<The sample table below is taken from an example TCK that includes both CLDC and a preverifier. These sample directories and files must be edited to match your installation.>

TABLE 4 [NAME TCK] directory contents

File or Directory	Contents
`javatest.jar`	JAR style archive for JavaTest software and JavaTest libraries. This file must be specified in the CLASSPATH shell environment variable in order to start the JavaTest software.
`testsuite.jtt`	File that contains information required by the JavaTest harness about the test suite.
`agent.jar`	Contains class and resource files for [AgentName] and for [NAME] test framework libraries.
`httpsrvr.jar` `httpclnt.jar`	J2ME only. Contains class files for implementations of the server and client.
<other JAR files as needed>	<Description>
`classes/`	Contains class files for the [NAME TCK].
`classes/preverified/`	Contains test classes that are preverified using the reference preverifier
`classes/shared/testClasses.lst`	This file contains the list of tests with their associated auxiliary files and classes needed for test bundling. It is required for running the tests and should not be modified.

TABLE 4 [NAME TCK] directory contents *(Continued)*

File or Directory	Contents
`doc/`	This directory and its subdirectories contain all of the documentation for the following: [NAME TCK], JavaTest, test APIs, (with the exception of `index.html` and `releaseNote.txt` files which are directly under the *TCK_DIRECTORY)*.
`doc/[name]-tck/`	Contains the [NAME TCK] User's Guide (`[name]-tck.pdf`)
`doc/javatest/`	Contains JavaTest-specific documentation: `javatest.pdf`: JavaTest User's Guide and Reference `tutorial.pdf`: JavaTest Tutorial `tutorial/index.html`: HTML version of the JavaTest Tutorial
`lib/`	Contains the following JavaTest files: environment files (`.jte`), parameter (`.jtp`) file, Exclude List (`.jtx`).
<source files>	<Descriptions and locations of source files.>
`src/share/classes/com/sun/cldc/communication/`	J2ME only. Contains source files for [NAME TCK] Test Framework interfaces and the HTTP implementation of these interfaces. See Appendix A, "{Implementing the Test Framework}" for details.
`tests/`	Contains all of the test program sources (`.java`) and test descriptions (`.html`) for the [NAME TCK]. The test hierarchy begins with the `testsuite.html` file in this directory (the JavaTest software refers to this file as the RootURL for the [NAME TCK]).
`tests/api/`	Contains all of the test program sources for the application programming interface (API) tests.

`solaris/bin/`
`win32/bin/`
`linux/bin/`
	OS related directories containing startup scripts for the JavaTest software (each script named `javatest`)

<Other directories as needed>	<Description>

There may be additional directories related to the JavaTest harness that are not found under the *TCK_DIRECTORY*. These are listed in the table below.

TABLE 5 User-defined [NAME TCK] directory contents

File or Directory	Contents
user_defined_work_dir	Work directories are user definable and are used to store test result files (`.jtr`).
user_defined_report_dir	The report directory is user-definable and is used to store the `harness.trace` file.

Installing the Agent

<The following example paragraphs assume that your TCK uses the standard JavaTest agent supplied with the JavaTest harness.>

The JavaTest Agent provided with the JavaTest harness is a lightweight program that uses a bidirectional serial connection supporting both TCP/IP and RS-232 protocols to communicate between the test system and the JavaTest harness. Other types of serial connections such as infrared, parallel, USB, and firewire connections can be added through the JavaTest API and modeled on the existing serial system.

The JavaTest harness includes an Agent Monitor window in the graphical user interface that you can use to control and monitor agents.

Installation of the JavaTest Agent is described in the *JavaTest User's Guide* and JavaTest online help.

<If your TCK uses a custom agent (named [AgentName] you must describe how it is installed on the system running the technology that is being tested. The example paragraph below is for a TCK which automatically downloads the agent.>

When running the [NAME TCK], the [AgentName] software is run on a target device where the [NAME] implementation under test is running. Its responsibility is to run the tests collected by the JavaTest harness and return the results of these tests via the particular communications route for the test setup.

The JavaTest harness includes an Agent Monitor window [p 64] in the graphical user interface that you can use to control and monitor agents.

The ([AgentName]) is automatically downloaded to the device being tested when the [NAME TCK] is run. (Assuming, of course, the device is properly connected to the system running the [NAME TCK].)

<If the custom agent is not automatically downloaded, or some user intervention is required to install it, the paragraph above should be replaced by a description of the agent installation steps.>

{Installing the Reference Implementation}

<Use this section for optional instructions on how to install the reference implementation. This could be full instructions or it could simply refer the reader to some other documentation source.>

CHAPTER 4

Starting and Configuring the JavaTest Harness

<This example chapter assumes that your TCK uses JavaTest harness version 3.x.>

This chapter describes basic [NAME TCK] execution and configuration. It contains the following sections:

- {Setting System Variables}
- Executing the JavaTest Harness Software
- JavaTest Harness Configuration
- {Special Set-Up Instructions}

In order for the [NAME TCK] to run, both system and JavaTest harness variables have to be properly set.

In essence, configuring the [NAME TCK] is a two part process:

1. Set your system variables as described in "{Setting System Variables}" below.
2. Set your JavaTest harness environment variables through the configuration interview as described in "JavaTest Harness Configuration" on page 33.

{Setting System Variables}

<If necessary, use this section to describe any system variables that need to be set.>

<Note that different engineers and QA people may personally use or prefer different system variable settings. If different people use different system variables, their command line syntaxes are going to be different. You need to determine which variables and syntaxes to document.>

Executing the JavaTest Harness Software

Before executing the JavaTest harness software, you must have a valid test suite and J2SE SDK 1.3 or greater installed on your system.

When executing the JavaTest harness, you can include arguments at the end of the command string that specify how it starts. These command-line options are described in your JavaTest documentation that is located at *<location>*.

Make sure that the correct `java` is in the execution path.

When you execute the JavaTest harness software for the first time, the JavaTest harness displays a Welcome dialog box that guides you through the initial startup configuration.

You can start the JavaTest harness in the following ways:

- **{Run a Startup Script**. You can use one of the startup scripts provided with the [NAME TCK] as described in "{Executing the JavaTest Harness—Scripts}" on page 32.}
- **Execute the JAR File**. If you have access to a command line, you can execute the harness from the top-level directory of your test suite by directly executing the JAR file at the command prompt: java `-jar lib/javatest.jar` *options*.
- **Double-Click the Icon (Windows platforms only)**. If you are using a GUI, your system may support double clicking the `javatest.jar` file icon to launch the harness.

When you execute the JavaTest harness for the first time it displays a Welcome to JavaTest dialog box.

- If it is able to open a test suite, the JavaTest harness displays a Welcome to JavaTest dialog box that guides you through the process of either opening an existing work directory or creating a new work directory as described in the JavaTest online help.
- If the JavaTest harness is unable to open a test suite, it displays a Welcome to JavaTest dialog box that guides you through the process of opening both a test suite and a work directory as described in the JavaTest documentation.

After you specify a work directory, you can use the Test Manager to configure and run tests as described in Chapter 6, "Testing Your Implementation."

{Executing the JavaTest Harness—Scripts}

```
<This sample section describes using optional platform-specific start-
up scripts that may (or may not) be supplied with your TCK.>
```

The [NAME TCK] provides <*N*> platform-specific startup scripts that can be used to execute the JavaTest harness (Solaris, Windows, and Linux) from the *TCK_DIRECTORY*. Each script is named `javatest` and is self-documented. They are found in their respective directories, as follows:

```
solaris/bin/javatest
win32/bin/javatest
linux/bin/javatest
```

JavaTest Harness Configuration

<This section assumes that your TCK is designed to provide JavaTest harness configuration information through a configuration editor interview. If you do not use an interview, or if for some reason you use the old JavaTest 2.x .jtp and .jte files rather than the configuration editor, you must write a different section to describe the process you use.>

In order for the JavaTest harness to execute the test suite, it requires information about how your computing environment is configured.

JavaTest Harness Configuration Overview

When you use the JavaTest harness, it checks for the configuration information that it needs. If configuration data is incorrect or missing, it prompts you to enter what is needed. This process is sometimes referred to as a "*configuration interview*" and the mechanism is called the "*Configuration Editor*."

Once the JavaTest harness GUI is displayed, you can run the Configuration Editor to change the configuration, or you can choose Run Tests > Start to begin a test run. When you start a test run, the JavaTest harness determines whether all of the required configuration information has been supplied:

- If the test configuration is complete, the test run starts immediately.
- If any required configuration information is missing, the JavaTest harness starts the Configuration Editor which displays a series of questions asking you for the necessary information. When you have finished entering the configuration data, you are asked if you wish to proceed with running the test.

At any point in the interview you can use the File > Save As menu item to save your test configuration to a `.jti` file. As explained in the JavaTest harness documentation, you can load a previously saved `.jti` file from the command line when you execute the JavaTest harness software. In that case, the JavaTest harness starts with the configuration settings specified in the `.jti` file.

Note – JavaTest harness configuration files (`.jti`) cannot be directly edited with a text editor. While the normal method of modifying a `.jti` file is to change settings in the JavaTest harness GUI and then saving the file, you can also use the `set` command in batch mode or the `editJTI` utility. Both are documented in the JavaTest harness help.

JavaTest Harness Configuration Editor

The JavaTest harness needs configuration information describing such things as the JavaTest harness host, the test device, and test options (such as which tests to run). To specify this information, you use the JavaTest Configuration Editor. The *JavaTest User's Guide* describes the Configuration Editor in detail.

To invoke the Configuration Editor, choose Configure > Edit Configuration in the Test Manager window. FIGURE 1 shows a typical Configuration Editor window.

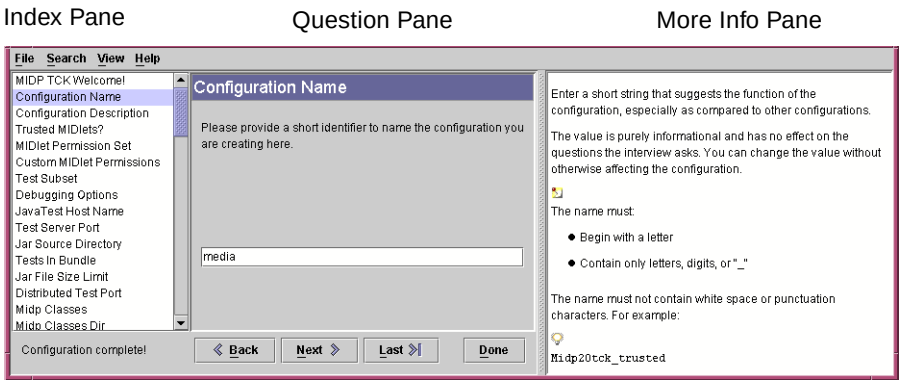

FIGURE 1 Typical Configuration Editor Window

The Configuration Editor asks you a series of questions, collectively called an "interview." It stores the answers in a configuration file (`.jti`).

You can have multiple configuration files, representing, for example, different test device configurations or option selections. You can load a configuration file by choosing Configure > Load Configuration in the Test Manager window or by specifying the configuration file name on the command line when you execute the JavaTest harness software. You can save the current configuration by choosing File > Save or Save As in the Configuration Editor window.

The Configuration Editor uses the answers in the current configuration to create the test environment. The test environment is a collection of environment entries that describe attributes of the test device, the JavaTest harness host, test options you have selected and so on. These entries control the execution of the test run that

commences when you choose Run Tests > Start in the Test Manager window. You can view the test environment by choosing Configure > Show Test Environment in the Test Manager window.

To use the Configuration Editor to create a new configuration, you proceed sequentially through the questions answering one, and then clicking the Next > button to see the next question. The following describes some important Configuration Editor features and behaviors:

- The More Info pane gives help for answering complex questions.
- When you have answered all questions in the interview, the Configuration Editor displays a message to that effect.
- The list of questions in the Index pane may change as you answer questions. For example, if you answer No to a question that asks if an optional test device feature is present, the interview will not ask you about attributes of that feature.
- The Configuration Editor hides some questions.
- The Configuration Editor validates some answers during the interview; if an answer is unacceptable, it will beep when you click Next >.
- You can return to a previously answered question by clicking its name in the Index pane.

For additional information, see:

- Appendix B, "Configuration Editor and Environment Variables," for a list of Configuration Editor questions and the environment variables they affect.
- "Configuration Failures" on page 65 for information on debugging test failures caused by incorrect environment values.

<If your TCK requires any additional specific configuration, add that information below. Keep in mind that the "More Information" feature of the interview should provide the necessary information.>

<If your TCK provides any default or template configuration files, describe them here and explain what site-specific parameters (such as hostname) have to be entered or changed.>

{Special Set-Up Instructions}

<Description of any other TCK-specific set up requirements or instructions required by your TCK. For example, starting a server for tests to communicate with.>

CHAPTER 5

Verifying the [NAME TCK]

This chapter describes how to start the [NAME TCK] and verify that it is properly configured. It contains the following sections:

Once the [NAME TCK] has been installed, Sun recommends that you set up and verify your test configurations in incremental steps in order to simplify any necessary troubleshooting. To do this, you first set up the JavaTest harness and the [AgentName] and pass a small series of tests.

[NAME TCK] Operating Assumptions

The following is assumed:

- J2SE SDK version 1.3 or later is installed on the system hosting the JavaTest harness.
- You are using JavaTest harness version 3.x.
- An implementation of [NAME] [*N.N*] is installed on the system hosting the JavaTest harness or on a system or device that the JavaTest host can access.
- <Any other required software (CLDC, for example) is installed.>
- {The device being tested meets the criteria listed in "{Components Provided by User}" on page 10. (The [NAME] Reference Implementation version 1.0.3 meets these requirements.)}
- <Any other assumptions or requirements>

{Using an Agent}

<Some TCK's use the standard JavaTest Agent supplied with the JavaTest harness, others use a custom agent. This optional section is for any instructions needed to run the agent.>

<For example, if your TCK uses the standard JavaTest Agent supplied with the JavaTest harness you could include the following:>

If the JavaTest Agent is used to run the tests for your product, you must start the agent before you begin the test run. Instructions on installing and starting the JavaTest Agent are provided in the *JavaTest User's Guide* and JavaTest online help.

If the JavaTest harness issues a request before the passive agent is started, the harness waits for an available agent until its timeout period ends. If the timeout period ends before an agent is available, the JavaTest harness reports an error for the test.

<If your TCK uses a custom agent (named [AgentName]) you must give the necessary instructions on how to start it. The instructions will, of course, be different for different agents. The example shown below is taken from a Sun TCK used to test the J2ME MIDP technology, your custom agent may require different instructions.>

Because the JavaTest agent will try to connect to a [AgentName] port when it is started, you should first choose Run Tests > Start from the JavaTest harness menu to start the built-in HTTP server before invoking the [AgentName].

The [AgentName] is run on the system or device running the [NAME] implementation that you wish to test. The method of starting the [AgentName] on a target device is implementation-specific. For example, for the Sun Reference Implementation, start the [AgentName] by running the following command. (The location of [name] must be set in the `PATH` variable.)

```
[name] -autotest http://JAVATEST_HOST:8000/test getNextApp.jad
```

Where:

- [name] starts `[name]`
- `-autotest` repeats after execution of the first JAR application.
- *JavaTest_Host* is the host on which the JavaTest harness is running.
- *8000* is the port on the host on which the JavaTest harness is running.

{See Appendix A, "{Implementing the Test Framework}" for further details.}

Verifying Installation and Setup

Once the [NAME TCK] has been installed, [Maintenance Lead] recommends that you set up and verify test configurations in incremental steps in order to simplify any necessary troubleshooting. [Maintenance Lead] recommends that you *first* set up the JavaTest harness {and the [AgentName]} and pass a small test to make sure that you have the correct JavaTest harness configuration and that the JavaTest harness {and agent} are running correctly.

The following general steps summarize this procedure:

1. **Set up and verify the JavaTest application installation.**

   Described in "Verifying JavaTest Harness Configuration {and Connection}" on page 39.

2. **Verify any other required components.**

   Described in "{Verifying Other Required Components}" on page 40.

Verifying JavaTest Harness Configuration {and Connection}

<The following example section assumes that your TCK is testing technology on a connected device using a custom agent named [AgentName]. This section is provided as an example, your situation may be quite different.>

To set up and verify correct JavaTest installation, perform steps below. These steps assume that the [Maintenance Lead] [NAME] <version NNN> has been installed on a device connected to the host running the JavaTest harness.

1. **Execute the JavaTest harness software.**

   "Executing the JavaTest Harness Software" on page 32.

2. **Choose Configure > Edit Configuration to open the Configuration Editor.**

   Provide configuration information appropriate for the [Maintenance Lead] [NAME] <version NNN> RI.

   See "{RI Configuration Editor Questions}" on page 41 for a list of Configuration Editor questions and appropriate answers.

3. **Establish your communications link to the RI.**

   If the RI is running on a different system other than the one running the JavaTest harness, make sure that both systems can communicate over the network using the HTTP protocol.

4. **Start the test run.**

   Choose Run Tests > Start to begin the test. If configuration information is incomplete, you will be asked to supply the missing data.

   Should you encounter any errors after clicking Start, refer to the JavaTest harness online help or the JavaTest User's Guide and Reference (located at *<directory_name>* in the [NAME TCK] distribution) for troubleshooting information.

Note – {When running the [NAME TCK], always start the JavaTest harness before starting the [AgentName] which occurs when you start the RI. For subsequent test runs, first exit the RI to release all used resources then invoke it again.}

---

5. **Start the Reference Implementation (RI).**

   On the system hosting the RI, issue the following command from the top-level RI home directory:

   <command line to start RI>

   Where:

`<Describe the command line elements.>`

   {Once the RI is started, it queries the JavaTest harness which then downloads and installs the [AgentName] on the system running the RI.}

   The JavaTest harness status bar grows while the JavaTest harness tracks statistics to record test progress.

6. **Check the results.**

   Check the test results as displayed by the JavaTest harness to make sure that everything is functioning correctly. Consult the JavaTest online help for information on test results.

7. **{(Optional) Terminate the [AgentName].}**

   {If necessary, provide instructions on how to terminate [AgentName]. For example, a reset or CTRL-C> or whatever other method is used.}

{Verifying Other Required Components}

`<Some technology implementations require additional software to be loaded on the device being tested. For example, a TCK might require the presence of Connected Limited Device Configuration (CLDC). This example section instructs the user to load and test such software.>`

A [NAME] implementation requires that a valid implementation of the Connected Limited Device Configuration (CLDC) be present on the [NAME] device.

- Before testing your [NAME] implementation on the target device, you must first install, set up, and test CLDC on the device. This mean that the CLDC implementation must pass the CLDC TCK tests.
- If you later make any changes to the CLDC implementation, you must re-run the CLDC TCK to confirm that the CLDC implementation on the device still passes all the required compatibility tests.

<Describe how to run the TCK that tests the required component. This might require referring the reader to the User Guide that describes the TCK used to test that component, or it might be that your TCK provides an internal method of invoking the needed tests in which case you should describe the necessary steps.>

{RI Configuration Editor Questions}

<If you think it useful, you can use a table like the one below to step the user through all of the Configuration Editor questions and answers appropriate for verifying the JavaTest harness configuration. The example shown below in TABLE 6 is taken from the MIDP User's Guide. Your questions and answers will be different.>

TABLE 6 lists the Configuration Editor questions and appropriate answers for verifying your configuration with the RI.

TABLE 6 Configuration Editor Questions for the RI

Item Name	Question	Answer/Action
Configuration Name	Please provide a short identifier for this configuration.	SampleConfiguration
Configuration Description	Please provide a short description of this configuration.	Sampleconfiguration
Trusted MIDlets?	Do you want to test untrusted or trusted MIDlet behavior in this run?	Select "Untrusted"
Custom MIDlet permissions	Which permissions will the test device/user grant to untrusted MIDlets?	Select nothing
Test subset	Which tests do you want to execute with this configuration?	Select "Automated"
Debugging options	Which MIDP TCK components would you like to have display debugging/verbose output?	Select all
Javatest host name	Please specify the name of the system on which you run the JavaTest Harness:	*JAVATEST_HOST*
Test Server Port	Please specify a port on the JavaTest host that the MIDP TCK's HTTP test server can use:	*JAVATEST_PORT*

TABLE 6 Configuration Editor Questions for the RI

Item Name	Question	Answer/Action
Jar Source Directory	Please specify a directory to hold Jar files:	Any temp directory: `C:\TEMP` or `/tmp`
Tests in Bundle	How many tests should the MIDP TCK attempt to pack into a Jar file?	Point to 1
Jar File Size Limit	Please specify a maximum size for Jar files:	60000
Distributed test port	Which JavaTest host port can the MIDP TCK use for coordinating distributed tests?	1908
Midp classes	Are the test device MIDP implementation class files in a directory or a Jar file?	Select “Directory”
Midp classes dir	Please specify the directory containing the MIDP implementation class files:	*MIDP_RI_DIR*\ `classes`
Color display	Does the test device display color or grayscale?	Select “Color”
Number of Colors	How many different colors can the test device display simultaneously?	256
Canvas Pointer Events	Does the test device support canvas pointer press/release events?	Select “Yes”
Canvas Pointer Motion Events	Does the test device support canvas pointer motion events?	Select “Yes”
Canvas Repeat Events	Does the test device generate repeated canvas events while a key is held down?	Select “No”
Double buffered Canvas	Are canvas graphics double buffered?	Select “Yes”
Text Box Capacity	What is the maximum number of characters that a text box can hold?	0
Text Field Capacity	What is the maximum number of characters that a text field can hold?	0
Foreground Color Value	Which color can the MIDP TCK use as a foreground color?	0
Foreground Color Name	What is the name of the foreground color?	Select “Black”
Background Color Value	Which color can the MIDP TCK use as a background color?	16777215
Background Color Name	What is the name of the background color?	Select “White”
Does the test device have a serial port?	Does the test device have a serial port?	Select “No”
Media Timeout	How long (in milliseconds) should tests wait for media connections?	30000
Volume Support For Tone	Does the test device's tone player support the VolumeControl interface?	Select “Yes”

TABLE 6 Configuration Editor Questions for the RI

Item Name	Question	Answer/Action
Sampled Sound Support	Does the test device support sampled audio playback?	Select “Yes”
Volume Support For Sampled Audio	Does the test device's sampled audio player support the VolumeControl interface?	Select “Yes”
Synthetic Sound Support	Does the test device support synthetic sound (MIDI)?	Select “No”
SecureSocketStreamSupport	Does the test device implement secure socket stream connections?	Select “Yes”
Secure Socket Type	Which protocol does the test device's SecureConnection implementation support?	Select “SSL”
Built-in Certificate?	Do you want the MIDP TCK to use its built-in certificate?	Select “Yes”
HTTP Server Port	Which JavaTest host port can TCK's HTTP server use?	8089
HTTPS Server Port	Which JavaTest host port can TCK's HTTPS server use?	7070
HTTPS Secure Protocol	Which protocol does the secure HTTP connection use?	Select “SSL”
Server Certificate File	Please generate a server certificate and specify the file it is located in:	your certificate
Server Certificate Alias	Enter the server certificate's alias:	your certificate's alias
Keystore Password	Enter the keystore's password:	keystore's password
Private Key Password	Enter the certificate's private key password:	your certificates' private key password
Secure Server Port	Which JavaTest host port can the TCK use to test secure connections?	8090
Socket Stream Support	Does the test device support socket stream connections?	Select “Yes”
ServerSocket Connection Support	Does the test device support incoming socket connections?	Select “Yes”
Outgoing Datagram Support	Does the test device support datagram transmission?	Select “Yes”
Incoming Datagram	Support Does the test device support incoming datagrams?	Select “Yes”
Free Device Port	Which test device port can the TCK use for optional connection testing?	50000
Push Registry Support	Does the test device support the push registry?	Select “Yes”

TABLE 6 Configuration Editor Questions for the RI

Item Name	Question	Answer/Action
Do you wish to run only selected sections of the test suite?	Do you wish to run only selected sections of the test suite?	Select “Yes”
Tests to Run	Specify the sections of the test suite you wish to run:	Select tree item: api javax_microedition lcdui gauge
Specify an Exclude List	Do you wish to specify an exclude list?	Select “Yes”
Which Exclude List	Which exclude list do you wish to use?	Select “Initial”
Specify Keywords?	Do you wish to specify a keyword expression to select / reject tests based on keywords contained in each test description?	Select “No”
Specify Status?	Do you wish to select tests to run based on their result in a previous run?	Select “No”
Time Factor	Specify a time factor that is applied to each test's default timeout. For example, specifying “2” doubles the time for each test (the default is 1):	1.0

CHAPTER 6

Testing Your Implementation

This chapter describes how to use the [NAME TCK] to test your implementation. It contains the following sections:

This chapter assumes that you have verified that your TCK is properly configured as described in Chapter 5, “Verifying the [NAME TCK].

Operating Assumptions—Testing a Product

<This section assumes that you are using JavaTest harness. Some of the example bullets below may not apply to your TCK, and your TCK may have assumptions not listed here.>

The following is assumed:

- That the JavaTest harness and the device being tested meet the criteria listed in “[NAME TCK] Specifications and Requirements” on page 4.
- J2SE JRE version *<x.x>* or later is installed on the system hosting the JavaTest harness.
- You are using JavaTest harness version *<x.x>* or later.
- That your [NAME] implementation has been installed on a target device, platform, or computational equivalent, which is accessible to the JavaTest harness host using the HTTP protocol.
- {Other required elements (CLDC, for example) are properly installed and configured.}

- {Any other requirements.}

Test Selection

Tests are selected for a given test run according to selection criteria that you specify. This section describes the different kinds of selection criteria you can use.

<This section assumes that your TCK uses JavaTest harness 3.x. If you are using some other test harness or version, the test-selection methods will vary.>

Test Selection Criteria

The selection criteria you specify can be thought of as a series of test filters that restricts which tests are run. At any given time, only tests that pass *all* of your selection criteria (all of your filters) are run.

There are different methods of specifying test-selection criteria:

Configuration Editor test-selection criteria. Some of the answers you give to Configuration Editor questions act as test-selection filters. The final standard value questions in the interview concern values related to test selection for the JavaTest harness. Those that affect test selection are:

<Describe the JavaTest harness 3.x test-selection standard value questions that apply to your TCK. The example bullets below are taken from the MIDP User Guide and are specific to MIDP 2.0. Since it is unlikely that they would apply to your TCK, they should be seen as examples only.>

- **Trusted versus Untrusted**: Answering “Trusted” or “Untrusted” to the Configuration Editor support for trusted MIDLets question, selects a subset of the [NAME TCK] tests.
  - If you answer “Untrusted” tests that contain the trusted keyword are filtered out and not run. (In other words, tests containing the keyword `untrusted` are run, and tests that contain neither `trusted` nor `untrusted` keywords are also run.)
  - If you answer “Trusted” no tests are filtered out. In essence, the `trusted` and `untrusted` keywords are ignored in terms of test selection. Tests are run regardless of whether or not they contain either keyword.
- **Permission set**: If you answer “Untrusted” to the Configuration Editor support trusted MIDLets question, you are asked to specify the permissions to be granted to the MIDlet suites. The permissions you grant act as a filter to select a subset of tests because a test will not run unless it has been granted (or denied) the appropriate permissions. In other words, a test may require that certain

permissions be granted, or it may require that certain permissions be denied; and the test will be run, or not run, according to which permissions you have granted or denied.

- **Test subset radio buttons**: The Configuration Editor asks you to select one of four radio-button keywords: automated, interactive, OTA, or all. The button you choose selects certain tests as follows:
  - `automated`: Tests that do not contain either the interactive or OTA keywords are selected.
  - `interactive`: Only tests containing the interactive keyword are selected.
  - `OTA`: Only tests containing the OTA keyword are selected.
  - `all`: Tests are selected regardless of whether or not they contain the interactive or OTA keywords.
- **Specify keywords**: The Configuration Editor allows you to specify a keyword expression. This is functionally equivalent to the JavaTest harness key word pane which is discussed in detail below. (Do not use this feature unless you fully understand its implications.)
- **Prior status**: The Configuration Editor allows you to select tests based on test status from a previous test run. This is functionally equivalent to the Prior Status tabbed pane which discussed below. Prior status selections chosen with this question replace any prior status selection specified through the tabbed pane, and vice versa. Note also that all previous criteria (restrictions) specified by the Configuration Editor and JavaTest harness GUI are applied to the tests you select by prior status. For example, if you chose the OTA radio button in the Configuration Editor and tests that failed with the Prior Status pane, only OTA tests that failed will be run.
- **Optional features**: The Configuration Editor asks you if various optional features are supported. Your answers act as a filter to select certain tests according to what is contained in a test's `selectif` field in the test's description.

JavaTest harness: You can also use the JavaTest harness GUI to specify test-selection criteria as follows:

- **JavaTest harness keyword field**: Additional test keywords can be entered in GUI's Keyword pane (standard view). Keywords that you enter here further restrict which tests are run. In other words, keywords entered in the Keyword pane are *in addition* to the filters set through the Configuration Editor except that keywords entered in this field *replace* any keywords specified through the Configuration Editor Specify Keywords question, and vice versa.

Note – Do not use the Keyword field unless you thoroughly understand how the keywords you enter will affect test selection and how keywords interact with test-selection criteria established with the Configuration Editor.

- **JavaTest harness Tests to Run**: You can use the Tests to Run pane in Standard Values view to select a subset of tests to run. Note, however, that all previous criteria (restrictions) specified by the Configuration Editor and JavaTest harness are applied to the tests you select by this method.
- **JavaTest harness prior status**: You can select tests using the Prior Status tabbed pane to select tests. You can select only those tests that failed, had an error, were not run, or passed in the previous test run. This is functionally equivalent to the Prior Status Configuration Editor question discussed above. Prior status selections chosen in this pane replace any prior status selection specified through the Configuration Editor Prior Status question, and vice versa.

Exclude List: Tests listed in the Exclude List are not run.

Note that you can ensure consistent test selection criteria for subsequent test runs by saving a particular combination of selection criteria in a JavaTest harness `.jti` file, and then loading that file from the command line when starting the JavaTest harness.

{Multiple Test Runs With Different Test Sets}

<Some TCKs are able to run all tests in a single test-run. Others require multiple test-runs with different test-selection criteria in order to correctly run all tests. This example section is for TCKs that require multiple test-runs. The example bullets below are taken from MIDP and may not apply to your TCK.>

As a general rule of thumb, in most cases it is not practical to run all necessary TCK tests in a single test-run.

- **Different security policies**. If the security policy of your device is configurable (different permissions can be granted or denied) then you should do multiple test runs. At a minimum, you should do one “trusted” run, and two “untrusted” runs (one with the device set to grant the maximum set of permissions and another run with the device set to grant the minimum permission set.)
- **Automated vs interactive**. Some tests run automatically while others require user intervention. If you try to run both automated and interactive tests in the same run, you will have to wait while automated tests are run in order to be present for those tests that require user intervention when they occur. This could be very time-consuming.

{Making Sure All Necessary Tests Have Been Run}

<This optional section is a companion section to ‘{Multiple Test Runs With Different Test Sets}’ above. It is intended for TCKs that require multiple test-runs with different test selection criteria.>

Because multiple test runs with different test-selection criteria are usually needed to completely test an implementation, it is important that care be taken to ensure that all necessary tests have been run. In other words, if you cannot run all necessary tests in a single run, you have to make sure that your combination of different test runs with different test selection criteria have covered all the required tests.

The JavaTest harness GUI's test-name color codes can be used to determine whether all needed tests have been run. The Test Tree Pane of the JavaTest harness GUI lists the [NAME TCK] tests by name. The test names are color-coded as follows:

- White: Test has not been run
- Grey: Test is excluded by test-selection criteria (including the Exclude List)
- Green: Test has been run and passed
- Red: Test has been run and failed
- Blue: There was an error in running the test

One method of ensuring that all tests have been run is to create different work directories based on primary test-selection criteria. In this context Primary test-selection criteria are the answers you provided to the Configuration Editor's Support trusted MIDlets and Permissions questions. (You can save your configuration in a `.jti` file that you can then use to set identical criteria in subsequent test runs.)

For each of these work directories you could:

- Do a single TCK run for that selection criteria (that is, a run that included all test subsets)
- Do several TCK runs with different secondary test-selection criteria such as keywords or Tests to Run folders. In this case after completing the different test runs you need to cancel the secondary test-selection criteria to see if any tests show up in white color (not-run).

The end goal is to make sure that at the end of your test cycle all required TCK tests are shown in green (run and passed) when all secondary test selection filters are turned off.

Using the [NAME TCK] to Test a Product

This section describes how to use the [NAME TCK] to test a [NAME] implementation. It contains the following sub-sections:

Running [NAME TCK] Tests—Basic Steps

To perform full testing of the [NAME] implementation on the target platform, device, or emulation, follow the steps listed below.

1. **Make sure that your test environment meets the basic requirements.**

   (See "Operating Assumptions—Testing a Product" on page 45 for details.)

2. **Make sure that your test environment configurations are correct.**

   (See Chapter 4, "Starting and Configuring the JavaTest Harness," and Chapter 5, "Verifying the [NAME TCK]" for details.)

3. **Provide a connection between your device, platform, or emulation and your PC or Workstation COM ports, if necessary.**

<Describe necessary details as appropriate.>

<The following two steps are examples that might (or might not) apply to implementations that have security-related aspects.>

4. **{Set the device's security mode, if necessary.}**

   {If the device's security mode has to be pre-set before a test run, do what is required to accomplish that.}

5. **{Set up your certificates.}**

   {Make sure that the necessary certificates are properly installed.}

6. **Launch the JavaTest harness software.**

   (See "Executing the JavaTest Harness Software" on page 32 for details.)

Note – {When running the [NAME TCK], always start the JavaTest harness before starting the implementation on your device which starts the [AgentName]. For subsequent test runs, first exit the [AgentName] to release all used resources and then invoke it again.}

---

7. **Select the tests to be run.**

<This example step assumes that you are using JavaTest harness 3.x.>

   Test selection is done through the JavaTest harness Configuration Editor and GUI as described in "Test Selection" on page 46.

8. **Start the test run.**

   Choose Run Tests > Start to begin the test run. If configuration information is incomplete, you are asked to supply the missing data. This also starts the Server which will now wait for the [AgentName] to request tests.

   The JavaTest harness tracks statistics relative to the files done, tests found, and tests done.

9. **Start the implementation on the target device under test in the manner appropriate for your platform.**

   The commands to start the implementation are device-specific. {(You may need to take security mode considerations into account when starting the device.) Depending on the tests you are running, you may need to start the [AgentName] in the manner appropriate for your implementation.}

10. **Check the results.**

    Test progress and results are displayed by the JavaTest harness. (See "Monitoring Test Results" on page 55 for details.)

<Some TCKs require that test suites be cleared from the implementation under test before starting a new test-run. Or the user may have to take steps to clear the agent. If so, describe them here.>

11. **{Completion step}**

    <If necessary, provide instructions on how to terminate the test-run. For example, a reset, or CTRL-C>, or whatever other method is used.>

{[Optional] Running Interactive Tests}

<Some TCKs have different kinds of tests. For example, interactive and non-interactive. If your TCK provides more than one kind of test, describe how to run them here. The example below describes how to run the interactive tests as used by the MIDP TCK. Your TCK may use different procedures.>

The [NAME] test suite consists of two kinds of tests:

- Non-interactive, which do not require user interaction to run.
- Interactive, which require some user input or other kind of interaction. The [NAME TCK] contains different kinds of interactive test that use different methods to report test results:
  - Tests that require users to perform some activity, such as viewing the results on the device screen, and judge a pass/fail result.
  - Tests where the test itself reports a pass/fail result, but still require user interaction.

The [NAME TCK]'s interactive tests that require the user to judge the pass/fail result differ according how the pass/fail judgement is made:

- **"YesNo" interface with reference screen shot images**. These kind of tests provide reference screen shot images to allow easy verification of the test result. You use these images to compare the results and make the final pass/fail judgement. FIGURE 2 shows a typical "YesNo" interface with reference images.

FIGURE 2 Interactive Test With Reference Images

- **"YesNo" interface without reference screen shot images**. These kind of tests do not provide reference screen shot images. Instead, the tests instructs you to perform some action, such as pressing a hardware key or tapping a touch screen, to produce a result. You make the pass/fail judgement by comparing the actual result to the expected result. FIGURE 3 presents a typical "YesNo" interface without reference images.

FIGURE 3 Interactive Test Without Reference Images

- **"Done" interface**. These tests are designed for verification of events generated by the device when your interact with it. The pass/fail judgement is automatically computed by the test. FIGURE 4 presents a typical "Done" interface type of test.

FIGURE 4 Done Interface Interactive Test

To run interactive tests, follow these steps:

1. **Read the test instructions.**

   The instructions are displayed in the top part of the window.

2. **Press the corresponding Test button.**

   There could be few test cases within a single test. Pressing each Test Button will execute the test code.

3. **Verify produced result.**

   Each test case also has a small description of the parameters used and the expected results. It is placed under each reference screen shot image.

   Note that:

   - A small number of tests will perform automatic result checks based on the generated events.
   - If the "No" button is pressed, a text field is presented which can be used to provide test failure details.

{Running Distributed TCK Tests}

<Some TCKs support distributed tests, others do not. This optional section is an example for those that do. This needs to be edited (or completely re-written) to meet the specifics of your TCK.>

The [NAME TCK] supports distributed tests for complicated APIs and for tests that are heavily dependent on external resources or designed to run on devices with constrained resources such as a small display. All data transfer is tested using distributed tests.

All communications in a distributed test framework are HTTP-based, so it is assumed that the default HTTP communication is properly implemented at this point as described in this chapter and in more detail in Appendix A, "{Implementing the Test Framework}."

There are two kinds of distributed tests:

- **Interactive Tests**: When your test environment is properly configured for distributed tests, the active agent ([AgentName]) running on the device under test executes the tests while the passive agent reacts to requests from you according to test instructions. In other words, an operator must be present during interactive tests to follow the testing prompts and instructions displayed on the passive agent.

  In this situation all test controls are moved to the passive agent. When you enter a command to create an object or to change a state of an object the result will be visible on the device's screen. As a general rule, almost all interactive tests utilize a "Yes / No" choice which you use to confirm a pass or fail condition. (There are a few tests for events that have "Done" choice.)

- **Automated Test**s: Automated distributed tests do not require any interaction from an operator.

When executing distributed tests, an agent will be automatically started in passive mode on the JavaTest harness host. The passive agent provides external resources and services (communication and OTA tests) or displays test prompts and controls for the user (interactive distributed tests).

Port Number for Passive Agent

By default, the passive agent uses port 1908.

To use a non-default passive agent port:

- **Specify a non-default port in the Configuration Editor.**

  When asked "`Which JavaTest host port can the [NAME TCK] use for coordinating distributed tests?`" in the Distributed Test Port question, specify the port you wish to use.

{Pre-installing Agent and Client Classes}

<Some TCKs support pre-installation of agent and client classes, others do not. This optional section is an example for those that do. This needs to be edited (or completely re-written) to meet the specifics of your TCK.>

By default, the [NAME TCK] includes the [AgentName] and a *<client>* implementation in each test bundle that is downloaded to the device being tested. You can reduce communication overhead by pre-installing one or both of these components on the test device and then directing the [NAME TCK] to not include the component(s) in the test bundles it downloads to the device.

{CLDC TCK tests can also pre-install the agent and client classes. You may use the same procedure for CLDC TCK that is described below for the [NAME TCK].}

Note – Sun's [NAME] Reference Implementation (RI) supports ROMizing classes into the KVM. In an example test case, ROMizing the [AgentName] and the *<client>* implementation added *<nn>*KB to the static memory footprint.

Pre-installing the [AgentName]

To pre-install the [AgentName], follow these steps:

1. **Install the classes into the test device.**

   The classes are contained in `[directory_name]/lib/agent.jar`. How you install the classes into the test device is determined by your implementation.

2. **Answer "Yes" to the "AdvancedFeatures -> Agent Preinstalled?" interview question.**

Pre-installing the *<client>*

To pre-install the default Client, follow these steps:

1. **Install the classes into the test device.**

   The classes are contained in `[directory_name]/lib/`*<filename>*`.jar`. How you install the classes into the test device is determined by your implementation.

2. **Answer "Yes" to the "AdvancedFeatures - > Client Preinstalled?" interview question.**

Monitoring Test Results

After the test run begins, you can track its progress using the test progress display fields, Progress Monitor dialog box, test tree, and information tabbed panes in the Test Manager window.

During and after a test-run, test names in the test tree are color-coded as follows:

- White: Test has not been run
- Grey: Test is excluded by test-selection criteria (including the Exclude List)
- Green: Test has been run and passed
- Red: Test has been run and failed
- Blue: There was an error in running the test

{If you are using the JavaTest Agent to run the tests on your system, use the Agent Monitor window to control and monitor the activity of the agent. Open the Agent Monitor window either by clicking its button in the toolbar or by selecting it from the Tools menu.}

{Test Export}

<Some TCKs support test export for running tests outside of the TCK, others do not. This optional section is an example for those that do.>

The Test Export feature allows TCK tests to be “exported” into `.jar` files with test classes and a minimal test framework for “stand-alone” execution. (Interactive, OTA and distributed tests cannot be exported using this feature.)

Note – This mode can be used *only* for QA or debugging purposes. It *cannot* be used for certification.

Exporting Tests in [NAME TCK]

To export tests, follow these steps:

1. **Go to the “Advanced Features...” question in the Configuration Editor and select “Yes”**
2. **The “Test export” question is displayed. Answer “Yes.”**
3. **Select Run tests->Start.**

The `.jar` files with exported tests are created in the directory that you specified in answer to the "Jar Source Directory" question. These Java files have names in the format `test`*N*`.jar` (were *N* is a number). For example, `test1.jar`, `test2.jar`, `test3.jar`.

Older files will be overwritten with new ones.

An HTML index file is created with the name `tcktests.html`.

Tests that are successfully exported are marked `Status.passed`.

Tests That Cannot be Exported

Tests that include "remote" components running on the JavaTest harness side cannot be exported for stand-alone execution. Attempts to export such tests will result in `Status.failed`.

Tests that cannot be exported include:

- Distributed Network tests
- Signature test
- Distributed Interactive tests
- {Other kinds of tests that cannot be exported}

Running Exported Tests

<This example is taken from MIDP. It will need to be re-written for your TCK.>

Exported Jar files can be used to run tests as follows (using the RI as an example):

```
bin\midp -classpath jar_file
    com.sun.tck.midp.javatest.agent.MIDletAgent
```

When run in the manner, tests print output and report their results to the console.

Note that the [NAME] [*N.N*] RI does not support relative URLs in HTML files. Therefore `tcktests.html` cannot be used for interactive browsing and running exported tests with the RI.

CLDC tests can also be exported. Note, however, that running "negative" CLDC tests should result in a failure (or abnormal VM termination). If a negative test results in `Status.passed`, that should be considered as a test failure. In this context, "negative" tests are those with the keyword `negative` in the CLDC TCK 1.0, and `cldc_typechecker_specific` in CLDC TCK 1.1

Producing Test Reports

After the test run is completed, you can use the JavaTest harness to create HTML reports for the test run. When you generate reports, you can specify the directory in which they are to be stored.

You can then open and view (or print) these report files with a web browser:

- The results are viewed with the JavaTest harness Test Browser.
- You can view the test results by going to the appropriate directory where the reports are stored and opening the test report contents list file `report.html` with any web browser.

Test result reports include the following kinds of information:

- Environment information—tester login, working directories, and other identification information
- Date and time for each event
- A log of the configuration interview questions and answers.
- List of tests or symbolic names that can be expanded to the exact list of the tests
- The list of excluded tests as specified by the Exclude List.
- A pass/fail report, including any return status, return values, or return message
- Any system messages, including exceptions and errors, generated by the tests.

In addition to test reports, the following report files are also generated:

- A `summary.txt` file in the `/report` directory that you can open in any text editor. The `summary.txt` file contains a list of all tests that were run, their test results, and their status messages.
- `passed.html` and `failed.html`: each lists test files by name, and in the case of failures, the reason for the failure.
- `error.html`: lists tests that neither pass nor fail but that have configuration problems, such as those which prohibit tests from launching.
- `env.html`: lists environment file variables fully evaluated.
- `excluded.html`: lists files excluded from the run.
- `config.html`: contains the configuration interview questions and answers.
- `report.html`: tallies the number of files passed or failed.

Individual test result files (`*.jtr`) are located in a hierarchical tree structure similar to the test suite itself. This means that individual result files are only overwritten when a given test is rerun. In other words, when one test is rerun only its test result file is overwritten, all other test result files are left unchanged. In this manner, you may change your test selection criteria and run groups of test at a finer granularity without affecting the test results of tests that are not being run.

CHAPTER 7

{Testing API Signatures}

<If your signature test is included in the test suite and run from within the test harness the same as other tests, you do not need to include this chapter. This chapter is only for situations in which the signature test needs to be run separately from the other tests.>

This chapter describes how to run a diagnostic test to compare the signatures of the public and protected methods, constructors, and fields in the jar file containing classes for the RI against an exact list of signatures from the API Specification. It contains the following sections:

Running the Static Signature test is required.

Overview

It is physically impossible to verify API signatures using the [Technology Name] implementation on the target device in the absence of reflection capabilities. However, API libraries which are burned into ROM or placed into the device in some other way, typically have prototypes that follow the standard class file format. In this case it is possible to explore the class files of the prototype to gain confidence that the API libraries that are actually present on the target device comply with the specification. The [NAME TCK] contains a tool called Static Signature test exactly for this purpose.

Running Signature Test

<Describe how to run the signature test provided with your TCK. Or, alternatively, refer readers to the appropriate Signature Test documentation. If there are configuration issues that need to be addressed, describe them here.>

CHAPTER 8

{Test-Specific Information}

<In many cases using the configuration editor interview is all the configuration a TCK needs. However, it may be that your TCK requires additional set up information for certain tests or classes of tests. This optional chapter is for describing that information. The sample material below is meant as an example of one approach, you will need to determine the best way to cover the necessary material.>

This chapter provides information required to configure, set up, and run specific [NAME TCK] tests.

This chapter is organized alphabetically according to major test categories. Each test section has three sections that describe information specific to the different types of tests:

- **Configuration**. The test configuration values required by each type of test.
- **Setup**. Special steps (if any) required to set up specific tests prior to running them.
- **Execution**. Special instructions (if any) required to execute specific tests.

<The example section below is intended as an illustration of one way to approach documenting specific test requirements. It is taken from the JCK User Guide.>

{Extra-Attribute Tests}

The table below shows one way to present area tested and attribute information.

TABLE 7 Example Area Tested-Attribute Table

Area tested	Additional attributes in class files
Test URL	vm/classfmt/atr

{Configuration}

The table below shows one way to present test configuration information.

TABLE 8 Example Configuration Information Table

Test Configuration Value	Description
`platform.nativeCodeSupported`	If your system supports native code, set this value to “true”. This value signifies that the JVM under test provides support for loading native code. If your system does not provide support for loading native code, set this value to “false”.
`PATH` `LD_LIBRARY_PATH`	The `jck-runtime.jte` file must include any necessary platform-specific variables to identify the path to the library file. On Windows 95/98/NT set the `PATH` variable, on Solaris set the `LD_LIBRARY_PATH` variable.

{Setup}

Because these tests use C code, it is not possible to define their compilation in a platform-independent way. Therefore, you must compile these tests before running them. These files must be compiled into a library named `jckatr` for loading using the method `System.loadLibrary(“jckatr”)`. To build the `jckatr` library, compile the file `jck/src/share/lib/atr/jckatr.c`, which contains a list of `#include` statements that refer to other files in the `jck/tests` directory.

When building the `jckatr` library, if the library is linked dynamically, you must set up a platform-dependent system environment variable such as `LD_LIBRARY_PATH` on Solaris, or PATH on Windows 95/98/NT in order to load the jckatr libraries.

The following instructions show how to build the `jckatr` library on Windows 95/98/NT and Solaris:

```
<The platform-specific descriptions are omitted from this example.
They can be found in the JCK Users Guide.>
```

{Execution}

No special requirements.

CHAPTER 9

Debugging Test Problems

There are a number of reasons that tests can fail to execute properly. This chapter provides some approaches for dealing with these failures. It contains the following sections:

Overview

The goal of a test run is for all tests in the test suite that are not filtered out to have passing results. If the root test suite folder contains tests with errors or failing results, you must troubleshoot and correct the cause to satisfactorily complete the test run.

- **Errors**. Tests with errors could not be executed by the JavaTest harness. These errors usually occur because the test environment is not properly configured.
- **Failures**. Tests that fail were executed but had failing results.

The Test Manager window provides you with a number of tools for effectively troubleshooting a test run.

Consult *JavaTest User's Guide* and JavaTest online help for detailed descriptions of the tools described in this chapter.

Test Tree

Use the test tree to identify specific folders and tests that had errors or failing results. Color codes are used to indicate status as follows:

- Green—Passed.
- Blue—Test Error.
- Red—Failed to pass test.
- White—Test not run
- Gray—Test filtered out (not run)

Folder Information

Click a folder in the test tree to display its tabbed pane.

Choose the Error and the Failed panes to view the lists of all tests in and under a folder that were not successfully run. You can double-click a test in the lists to view its test information.

Test Information

To display information about a test, click its icon in the test tree or double-click its name in a folder status pane. The tabbed pane contains detailed information about the test run and, at the bottom of the pane, a brief status message identifying the type of failure or error. This message may be sufficient for you to identify the cause of the error or failure.

If you need more information to identify the cause of the error or failure, use the following panes listed in order of importance:

- Test Run Messages contains a Message list and a Message pane that display the messages produced during the test run.
- Test Run Details contains a two column table of name/value pairs recorded when the test was run.
- Configuration contains a two column table of the test environment name/value pairs derived from the configuration data actually used to run the test.

Agent Monitor

The JavaTest harness includes an Agent Monitor window in the graphical user interface that you can use to control and monitor agents. This tool can be used to see if you have made a connection to a JavaTest Agent(s). You can start the Agent Pool (for Active Agents) using the listening checkbox.

Debugging Option

You can turn on or off the debugging output for TCK components by answering the "Debugging Options" question in the Configuration Editor. Turning on debugging produces extensive debugging information in the implementation's standard output.

Report Files

Report files are another good source of troubleshooting information. You may view the individual test results of a batch run in the JavaTest Summary window.

Another important resource for debugging test failures is the work directory report called `failed.html`.

Configuration Failures

In some cases, configuration failures are easily recognized because many tests fail the same way. If all your tests begin to fail, you may want to stop the run immediately and start viewing individual test output to determine which environment variable is incorrect. (Note, however, that in the case of full-scale launching problems where no tests are actually processed, report files are usually not created.)

In other cases, a single test, or small set of tests, may fail because of an incorrect environment setting. In other words, a test failed because an incorrect environment value was passed to it. You can sometimes determine the name of the environment entry from the test description or the test run messages. To do this, select the test in the JavaTest test tree, then click the Test Description or Test Run Messages tab.

TABLE 9 on page 80 shows the source of environment most value settings. Most values come from Configuration Editor interview answers. After correcting an incorrect Configuration Editor answer, the test then runs properly with the right environment settings.

<The example below is taken from MIDP, you may want to provide a similar example from your TCK.>

For example, suppose the test `api/javax_microedition/lcdui/Canvas/index.html#IsDoubleBuffered` fails. The Test Description's `executeArgs` field shows that the test takes an argument named `CanvasisDoubleBuffered`. Choosing JavaTest Configure > Show Test Environment reveals that the value of `CanvasisDoubleBuffered` is false. However, that is an error; the value should be true. Looking up `CanvasisDoubleBuffered` in TABLE 9 on page 80 shows that value can be changed by changing the answer to the Double Buffered Canvas interview question.

TABLE 9 on page 80 also gives the JavaTest interview question tag of each entry whose value comes from an interview question. You can use this name to temporarily set the value of an entry from the command line with the `-set` option. For example, you can set the `httpPort` entry to 9999 as follows:

```
java com.sun.javatest.tool.Main -batch -open ../default.jti
    -set midptck.vm.httpPort 9999 -runtests
```

Hostnames and DHPC

<This section may not apply to all TCKs. Check with your engineers to see if it is applicable.?

For machines that use DHPC, you must use the correct DNS name for the Computer Name. Error messages such as: `javax.microedition.io.ConnectionNotFoundException: TCP open` may indicate an incorrect DNS name.

For example, on a Windows 2000 machine under Settings > Control Panel > System > Network Identification > Properties you can enter a "Computer name." Not any name can be entered, it must be the proper DNS name.

You should be able to access this machine over the network using this DNS name. This DNS name should be used in the Configuration Editor to answer the "JavaTest Host Name" question.

CHAPTER 10

{Product-Specific Chapter Template}

<This chapter template file can be used for any additional product-specific chapters that you require. For FrameMaker users, it already contains the variables and conditional text definitions used in other chapters.>

APPENDIX **A**

{Implementing the Test Framework}

<Many TCKs require the user to provide necessary software components that the TCK requires in order to function. For example, a user might need to create a communications channel, or Application Management Software (AMS). Use this appendix to provide users with the information they need to create whatever they have to provide. The example content below is adapted from the MMA 1.1 TCK. Your TCK may have quite different requirements.>

This appendix describes the test framework design and user-created components required by your TCK. It contains the following sections:

The [NAME TCK] allows implementation of platforms for a wide range of devices with different types of communication capabilities and different internal representation formats for applications.

An implementation of the [NAME] [VersionNumber] specification requires some underlying software that conforms to other Java specifications as follows:

- **MIDP plus CLDC**. Conformance to the Mobile Independent Device Profile (MIDP) 1.0 or 2.0 plus CLDC 1.0 or 1.1 specifications as appropriate. In this case, the [NAME TCK] assumes that your MMAPI implementation includes valid implementations of both the MIDP and CLDC specifications that have been tested by the appropriate MIDP TCK.

  Note—Since the MIDP implementation provides an HTTP communications protocol, the communications channel implementation described in this appendix is not necessary. In other words, if your [NAME] implementation conforms to the MIDP specification this Appendix does not apply and can be ignored.

- **CLDC plus IllegalStateException**. Connected Limited Device Configuration (CLDC) 1.0 or 1.1 plus a profile that includes the IllegalStateException. In this case, the [NAME TCK] assumes that your MMAPI implementation includes a valid implementations of the CLDC specification plus some kind of profile the provides the IllegalStateException as tested by the appropriate CLDC TCK.

  In this case, you may have to implement the test framework as described in this Appendix.

Testware Components

The JavaTest harness runs on a PC/Workstation. Its responsibility is to package groups of tests into jar files (test bundles) and make them available to the target device running the implementation under test.

JavaTest harness uses its server to perform the following functions:

- To send test bundles to the target device (including the test [AgentName]).
- To convert them to the application format of the target device, if necessary.
- To dispatch tests for execution and receive test results.

The [NAME TCK] provides the CLDCAgent for implementations the conform to a CLDC specification plus a profile that provides the IllegalStateException. The agent is pre-packaged with each test bundle. Its task is to execute tests from this bundle and to report results to the JavaTest harness. The agent uses the client to communicate with the JavaTest harness.

Additionally some Application Management Software (AMS) is required to be present on the target device. Its task is to repeatedly download test bundles from the JavaTest harness and to execute them. (This software is sometimes referred to as a Java Application Manager or AMS.) Due to significant variations among potential devices on which the [NAME] might run, the details of application management are highly device-specific and implementation-dependent. A typical implementation of this functionality is a native application used to download, store and execute Java applications. For simplicity, the term “AMS” will be used throughout this document to refer to this functionality.

The server and the client are Java implementations of the interfaces defined in the [NAME TCK]. The AMS component is typically written in native code.

When to Plug In Your Own Implementation and What to Plug In

By default, implementations of the communication channel for CLDC are assumed to meet the following criteria:

- The server uses the HTTP protocol to communicate with the agent and the AMS. It is implemented using the `java.net socket` API.
- The client uses the HTTP protocol to communicate with the server. It is implemented using the GenericConnection Framework APIs provided with the Reference Implementation (RI).
- Both the server and the AMS use application descriptor data to transfer applications over HTTP. The format of the descriptor is defined in "The Default Implementation of the Communication Channel" on page 75.
- The target device uses the standard JAR file format to download and store applications.
- The CLDC-based runtime environment in the device uses the standard Java application model with the `main()` method as an entry point.

Therefore, if your implementation does not support HTTP, you need to provide at least your own implementation of the server and the client. However, even if your implementation does support HTTP, in some cases you may have to provide your own versions of them.

For example, if your [NAME] implementation uses other APIs to implement HTTP, the client needs to be rewritten. Similarly, if your implementation of the AMS does not understand the application descriptors generated by the HTTP server, or it uses a format other than JAR to download and store applications, then the HTTP server needs to be rewritten. It may also have to be rewritten to provide a Java wrapper for CldcAgent to make it runnable on CLDC-based runtime environments with a different application model. (Although there is another way around this, as follows: the AMS may be modified to support the default descriptor format or JAR application format.)

Details are discussed further in "The Default Implementation of the Communication Channel" on page 75.

Communication Channel Components

The communication channel may be viewed from the perspective of three components: the client, the server, and the AMS. These components are discussed below.

Client

On the device/agent side, the following interface is used:

```
public interface Client {
/**
* Initialization
*/
void init(String[] args);
/**
* Reads next test.
* Returns null if all tests from this application have already
* been executed. In other words, null is an exit signal to the
* agent application.
*/
byte[] getNextTest();
/**
* Sends test result.
*/
void sendTestResult(byte[] res);
}
```

This interface allows for any test runner to receive execution requests and send execution results in a form of a byte array. The implementation of this interface should have a no-args constructor, and the actual initialization is performed in the `init()` method.

The client may assume that it is accessed by only one thread at a time (agent thread), so it doesn't have to be thread safe.

Server

On the JavaTest harness PC/Workstation-side, in addition to the server, the concept of the *Test Provider* is introduced. The Test Provider functions as a server to the server itself. The server knows its Test Provider and calls its methods in order to pass the data from the client to the Test Provider or vice versa.

Note that the server has no test related logic inside, all it does is just forward data. It is as lightweight as possible.

The definition of the interfaces is as follows:

```
public interface server {
/**
* Initialization
*/
void init(String[] args);
/**
* Starting
*/
void start();
```

```
/**
* Stopping
*/
void stop();
/**
* Set test provider. There is only one test provider
* per server. Next call to setTestProvider removes the previous
* one. Null argument causes removal of current test provider.
*/
void setTestProvider(TestProvider tp);
/**
* Returns current test provider.
*/
TestProvider getTestProvider();
}
public interface TestProvider {
/**
* Returns main application class for this test provider.
* The name is the same for all applications generated
* by this test provider.
*/
String getAppMainClass();
/**
* Returns the directory in which application jar files
* are stored.
*/
String getJarSourceDirectory();
/**
* Reads next test.
* Returns null if all tests from this application have already
* been executed. In other words, null is an exit signal to the
* test runner application.
*/
byte[] getNextTest();
/**
* Sends test result.
*/
void sendTestResult(byte[] res);
/**
* Returns next application to execute. This is a file name
* relative to the directory returned by getJarSourceDirectory().
*
* May return null if no application is currently available.
* In this case JAM is expected to repeat request some time later.
*/
String getNextApp();
}
```

The server is supposed to run in a separate thread. As with the agent, it should have a no-args constructor, and the initialization is performed in the `init()` method.

The server assumes that there are single instances of both the client and AMS contacting it. Parallel test execution is not supported.

AMS

The communication between the server and the AMS is implementation specific. On a high abstraction level, the device can send only one command to the server, which is `getNextApp`.

On the other hand, the server should be able to respond in three different ways, as follows:

- `OKAY +` *application descriptor*: if `TestProvider.getNextApp()` returned non-empty string
- `RETRY`, possibly with delay time: if `TestProvider.getNextApp()` returned empty string
- `DONE`: if `TestProvider.getNextApp()` returned null.

For example, assume that the target device has an AMS that is augmented with the required functionality, termed the test mode. When the AMS is running in test mode, it performs the following steps:

1. Contacts the server with the `getNextApp` command.
2. The server response may be one of the following:
   a. `OKAY +` *application descriptor.* AMS downloads and executes the application, and then returns to Step 1.
   b. `RETRY` command (if no application is currently available). The AMS waits for some period of time and then returns to Step 1. The exact period of wait time is either received from the server with this command or it may be a AMS property.
   c. `DONE` command (if no more applications are expected). AMS may either exit the loop or behave similarly to 2b.

Note – In an optimized scenario, the AMS might cache the last downloaded application and reuse it.

The Default Implementation of the Communication Channel

The HTTP protocol is used for communications. Both GET and POST requests must be supported. There is some designated location, or system area, through which all test requests/responses should go. The only requirement is that all communications via this area should come through uncached.

Client

The client is initialized with a single String argument containing the URL of the system area, for example:

```
http://server:8080/SYSTEM/
```

`getNextTest()` is implemented through HTTP GET, for example:

```
GET http://server:8080/SYSTEM/getNextTest HTTP/1.0
```

`sendTestResult()` is implemented through HTTP POST, for example:

```
POST http://server:8080/SYSTEM/sendTestResult HTTP/1.0
```

Server

Generally, HTTP implementation of the server should work as a regular HTTP server. The directory returned by `TestProvider.getJarSourceDirectory()` functions as the server root directory. Requests that are addressed to the system

area are handled in a special way. Specifically, the following actions are implemented for each input. Also see "The Default Implementation of the Communication Channel" on page 75 for additional details

`getNextTest`

The server calls `TestProvider.getNextTest()`. If it returns non-null value, the server replies with `Status 200 (OKAY)` and sends the content of the returned array as the body. Otherwise the server replies with `Status 404 (Object Not Found)`.

`postTestResult`

The server collects the data passed by the client into the byte array and passes it to the Test Provider calling `TestProvider.sendTestResult()`. Also `Status 200 (OKAY)` is returned to the client.

`getNextApp`

The server calls `TestProvider.getNextApp()`. If it returns null, the server replies with `Status 404 (Object Not Found)`. If it returns an empty string, the server replies with `Status 503 (Object Unavailable)` and adds the following line to the response header:

```
Retry-After: #
```

where # is a number of seconds to wait before contacting the server again. If the call to `TestProvider.getNextApp()` returns a non-empty string, the server replies with `Status 200 (OKAY)`, adds the following response header:

```
Content-Type: application/x-jam
```

It then sends the following application descriptor in the body:

```
Application-Name: Test Suite
JAR-File-URL: http://<httpHost>:<httpPort>/
<TestProvider.getNextApp()>
JAR-File-Size: <the size of the test bundle>
Main-Class: <TestProvider.getAppMainClass()>
Use-Once: yes
```

AMS

The AMS is able to send the `getNextApp` request to the server, for example:

```
GET http://server:8080/SYSTEM/getNextApp HTTP/1.0
```

Depending on the status code of the response, it either exits (if code `404` is received) or retries after the specified time (if code `503` is received), or does the following for response code `200`:

- Reads and parses the application descriptor
- Sends a `GET` *JAR-File-URL* request to the server and downloads the test bundle
- Executes the downloaded bundle using *Main-Class* as the main class

Plugging In Other Implementations of Server, Client, and AMS

The integration of the AMS with the implementation under test is highly device specific and therefore out of the scope of this document. For the server and the client, you need to answer the Configuration Editor questions so that they refer to your implementations of them.

For the server you should specify the following:

- <List the appropriate Configuration Editor questions and answers>

For the client, you should specify the following:

- <List the appropriate Configuration Editor questions and answers>

Note that the implementation of the client must be packaged into a single JAR file. On the other hand, the location of the server classes should follow the standard Java CLASSPATH conventions.

APPENDIX **B**

Configuration Editor and Environment Variables

<Note that the tables below are those for the MIDP TCK. They are provided here as one example of how you might choose to present this information. Since most configuration interviews are created separately for each TCK, most of your TCK Configuration Editor questions and associated variables will be quite different than those shown below. You need to have the TCK engineer responsible for creating the interview provide you with the information that applies to your TCK.>

This appendix describes the environment variables set by different Configuration Editor questions.

- TABLE 9 lists the environment variables and corresponding Configuration Editor questions used in tests of a MIDP implementation.
- {TABLE 10 lists the environment variables and corresponding Configuration Editor questions used in tests of CLDC running on a MIDP device.}

Note that not all environment entry values can be changed:

- An entry whose source is denoted as (constant) in TABLE 9 {or TABLE 10} is determined by the [NAME TCK] and cannot be changed.

- An entry value whose source is denoted as (computed) in TABLE 9 {or TABLE 10} may change if you change the value of a related entry whose source is an interview question.

TABLE 9 [NAME TCK] Environment Entry Sources

Environment Entry	Interview Question Name or Other Source Interview Question Tag
`agent`	(constant) (none)
`agentJar`	(constant) (none)
`badURL`	(constant) (none)
`badURL2`	(constant) (none)
`badURL3`	(constant) (none)
`CanvashasPointerEvents`	Canvas Pointer Events midptck.toolkit.canvasPointer
`CanvashasPointerMotionEvents`	Canvas Pointer Motion Events midptck.toolkit.canvasMotion
`CanvashasRepeat`	Canvas Repeat Events midptck.toolkit.canvasRepeat
`CanvasisDoubleBuffered`	Double Buffered Canvas midptck.toolkit.canvasDb
`classpath`	(constant) (none)
`CldcTCKAgent`	(constant) (none)
`CldcTCKCommand`	(constant) (none)
`client`	(constant) (none)
`clientJar`	(constant) (none)
`Color1`	Foreground Color Value midptck.toolkit.fgVal
`Color1Name`	Foreground Color Name midptck.toolkit.fgName
`Color2`	Background Color Value midptck.toolkit.bgVal

TABLE 9 [NAME TCK] Environment Entry Sources *(Continued)*

Environment Entry	Interview Question Name or Other Source Interview Question Tag
`Color2Name`	Background Color Name midptck.toolkit.bgName
`command.testExecute`	(constant) (none)
`description`	Configuration Description midptck.envDesc
`httpHost`	JavaTest Host Name midptck.vm.httpHost
`httpPort`	Test Server Port midptck.vm.httpPort
`httpServerPort`	OTA Port midptck.otaPort
`isColor`	Color Display midptck.toolkit.isColor
`isURLS`	(constant), JavaTest Host Name, Test Server Port, (constant) (none)
`jarFileSizeLimit`	Jar File Size Limit midptck.vm.jarFileSizeLimit
`jarSourceDir`	Jar Source Directory midptck.vm.jarSourceDir
jks.signer	Custom Signing Class or (constant) if Custom Signing Class = No midptck.trusted.signerClass
jks.signer.args	Built-in Certificate? midptck.trusted.defaultPki
`maxSizeTextBox`	Text Box Capacity midptck.toolkit.tbSize
`maxSizeTextField`	Text Field Capacity midptck.toolkit.tfSize
`media.WavOrMidi`	Sampled Sound Support midptck.commmedia.wave
`media.WavOrMidi`	Synthetic Sound Support midptck.commmedia.midi
`mediaTimeout`	Media Timeout midptck.commmedia.timeout
`midpClasses`	Midp Classes Dir or Midp Classes Jar midptck.midpClassesDir or midptck.midpClassesJar
network.comm.baudRate	Optional Baud Rate midptck.serialport.baud

TABLE 9 [NAME TCK] Environment Entry Sources *(Continued)*

Environment Entry	Interview Question Name or Other Source Interview Question Tag
network.comm.midCOM	Test Device Serial Port midptck.serialport.midPort
network.comm.remoteCOM	JavaTest Host Serial Port idptck.serialport.agentPort
network.datagram	Outgoing Datagram Support midptck.connection.datagOut
network.datagramreceiver	Incoming Datagram Support midptck.connection.datagIn
network.https.certFile	Server Certificate File midptck.connection.certFile
network.https.secureProtocol	HTTPS Secure Protocol midptck.connection.httpsProto
network.MIDHost	Device Host midptck.connection.devHost
network.MIDPort	Free Device Port midptck.connection.freePort
network.push	Push Registry Support midptck.connection.pushReg
network.push.alarmlaunch	Push Alarms midptck.connection.pushAlarm
network.push.datagramreceiver	Push Transports midptck.connection.pushProtos
network.push.serversocket	Push Transports midptck.connection.pushProtos
network.securesocket	Secure Socket Stream Support midptck.connection.secureSoc
network.securesocket.connProtocol	Secure Socket Type midptck.connection.secureProto
network.securesocket.keypass	Private Key Password or (constant) midptck.connection.keyPass
network.securesocket.keystore	Keystore or (constant) midptck.connection.keyStore
network.securesocket.port	Secure Server Port midptck.connection.securePort
network.securesocket.storepass	Keystore Password or (constant) midptck.connection.storePass
network.serversocket	ServerSocketConnection Support midptck.connection.socketIn

TABLE 9 [NAME TCK] Environment Entry Sources *(Continued)*

Environment Entry	Interview Question Name or Other Source Interview Question Tag
network.shutdownTime	MIDlet Recycle Time midptck.connection.MIDletRecycle
network.socket	Socket Stream Support midptck.connection.socketOut
numColors	Number of Grays or Number of Colors midptck.toolkit.numGrey or midptck.toolkit.numColors
`OTAHandlerArgs`	OTA Class Arguments or (constant) if Custom OTA Class = No midptck.otaa
`OTAHandlerClass`	OTA Class (or constant if Custom OTA Class = No) midptck.customOtah
`passivePort`	Distributed Test Port midptck.passivePort
`platformPermissionRestrictions`	Custom Midlet Permissions midptck.customPerms
`product`	(constant) (none)
`remote.networkAgent`	(computed) midptck.Parameters
`remoteVerbose`	Debugging Options (Distributed Tests) midptck.verbose
`server`	(computed), Debugging Options (Test Server) (none)
`serverJar`	(constant) (none)
`serverURL`	JavaTest Host Name, Test Server Port (none)
`signerJar`	(constant) (none)
`SupportServer`	(constant), Debugging Options (HTTP Server), JavaTest Host Name, HTTP Server Port midptck.connection.httpPort
`SupportServer1`	(constant), Debugging Options (HTTPS Server), JavaTest Host Name, HTTPS Server Port midptck.connection.httpsPort
`testURLS`	(constant), JavaTest Host Name, Test Server Port midptck.commmedia.midiVolume

`<The optional TABLE 10 below is for those TCKs that require CLDC.>`

{When you test a CLDC implementation with the [NAME TCK], the test environment contains a subset of the [NAME TCK] entries and some additional entries that the [NAME TCK] environment does not have. TABLE 10 lists the CLDC environment entries, their question tags, and the sources of their values. }

TABLE 10 MIDP+CLDC TCK Environment Entry Sources

Environment Entry	Interview Question Name or Other Source Interview Question Tag
`agent`	(constant) (none)
`agentJar`	Agent Jar `midpcldctck.vm.advanced.agentJar`
`classpath`	(constant) (none)
`CldcTCKAgent`	(constant) (none)
`CldcTCKCommand`	(constant) (none)
`client`	Client Class or (constant) if Advanced Features = No `midpcldctck.vm.advanced.clientClass`
`clientJar`	Client Jar or (constant) if Advanced Features = No `midpcldctck.vm.advanced.clientJar`
`command.testExecute`	(constant) (none)
`description`	Configuration Description midpcldctck.confDesc
`httpHost`	JavaTest Host Name midpcldctck.vm.httpHost
`httpPort`	Test Server Port midpcldctck.vm.httpPort
`httpServerPort`	OTA Port midptck.otaPort
`jarFileSizeLimit`	Jar File Size Limit midpcldctck.vm.jarFileSizeLimit
`jarSourceDir`	Jar Source Directory midpcldctck.vm.jarSourceDir
`server`	(computed), Debugging Options (Test Server) or (constant) if Advanced Features = No `midpcldctck.vm.advanced.`serverClass
`serverJar`	Server Jar or (constant) if Advanced Features = No `midpcldctck.vm.advanced.serverJar`
`serverURL`	JavaTest Host Name, Test Server Port `midpcldctck.vm.advanced.serverURL`

APPENDIX C

Exclusion Lists and Result Files

<This example Appendix assumes you are using JavaTest harness 3.x or later. If you are using some other test harness, re-write this appendix as necessary.>

This appendix describes the JavaTest harness exclusion lists and result files. It covers the following topics:

Exclude List Files

JavaTest harness exclude list files (`.jtx`) are used within theJavaTest harness to omit one or more tests from being run.

Note – You should regularly check the Java Partner Engineering web site (`javapartner.sun.com`) for updates to the exclude list.

The [NAME TCK] contains multiple exclude list files (`.jtx`) that specify those tests that are known to be invalid; therefore, these tests are not required to be successful or even to be run using a [NAME] system configuration. There are separate exclude lists for each of the [NAME TCK] component test suites. In other words, there is one exclude list for the MIDP test suite, and another exclude list for the [NAME] test suite, and so on. Each exclude list is stored in the `/lib` directory of its corresponding test suite.

Exclude list files have file names based on the corresponding test suite version numbers. For example: `[name]-tck_10.jtx` for the [NAME TCK] 1.0 test suite.

Note – From time to time, updates to exclude lists are made available on the javapartner.sun.com web site. You should always make sure you are using an up-to-date copy of each exclude list before running the [NAME] to verify your implementation.

A test might be added to the exclude list for reasons such as:

- An error in a [Maintenance Lead] Reference Implementation that does not allow the test to execute properly has been discovered.
- An error in the specification that was used as the basis of the test has been discovered.
- An error in the test has been discovered.

Note – Exclusion lists are the proprietary property of [Maintenance Lead] and are **never** to be modified by a Licensee.

The Exclude List you specify with the JavaTest harness Configuration Editor must point to the exclude list file for your particular version of the [NAME TCK].

Exclude File Format

The exclude list is a text file, where each line represents a single entry. The general format for a JavaTest exclude list file entry is as follows:

- Relative Test URL
- White space
- Bug numbers separated by commas
- White space
- Platforms separated by commas
- White space
- Comments
  - Comments to the whole exclude list appear on lines starting with #
  - Comments to for a particular entry start with a # and end with a space

Note – The comments are used when JavaTest harness generates test run reports.

An example of this file format is as follows:

```
api/javax_microedition/lcdui/Font/index.html#CharsWidth[Font2007]
    4370531 SPEC
api/javax_microedition/lcdui/interactive/DateField/
    index.html#DateField          4478529   RI
api/javax_microedition/lcdui/interactive/ImageItem/
    index.html#SetAltText[ImageItem2003]         4367759   SPEC
```

```
api/java_util/TimerTask/index.html#ScheduledETime[TimerTask2008]
   4501263   TCK
```

Result File Format

The information that is presented in the Test Summary Browser is also written to JavaTest result files.

These result files are stored in the work directory that you specified with the Configuration Editor or the JavaTest harness GUI. JavaTest harness creates a test result hierarchy similar to the test suite hierarchy that contains your tests. Result files for the tests in your test run appear under the appropriate directory within this result hierarchy.

An example of this file format is as follows:

```
#Test Results (version 2)
#Wed Sep 05 18:25:58 PDT 2001
#checksum:47294a2ccbd86a54
#-----testdescription-----
$file=your-tck-workdir/tests/api/javax_microedition/lcdui/Canvas/
   index.html
$root=your-tck-workdir/tests
executeClass=javasoft.sqe.tests.api.javax.microedition.lcdui.Canva
   s.GetGameActionTests
id=GetGameAction
keywords=positive runtime
source=GetGameActionTests.java CvGen.java
title=Tests for public int getGameAction(int keyCode)

#-----environment-----
CldcTCKCommand=com.sun.tck.cldc.javatest.CldcTCKCommand
command.testExecute=$CldcTCKCommand $testPath $testExecuteClass
   $testExecuteArgs

#-----testresult-----
description=file:/your-tck-workdir/tests/api/javax_microedition/lcdui/
   Canvas/i
ndex.html#GetGameAction
end=Wed Sep 05 18:25:58 PDT 2001
environment=[name]-tck
execStatus=Passed. tests:2; passed:2
javatestOS=SunOS 5.7 (sparc)
javatestVersion=2.1.6
script=com.sun.jck.lib.JCKScript
sections=script_messages testExecute
start=Wed Sep 05 18:25:52 PDT 2001
status=Passed. tests:2; passed:2
test=api/javax_microedition/lcdui/Canvas/index.html#GetGameAction
```

```
timeoutSeconds=600
work=your-tck-workdir/api/javax_microedition/lcdui/Canvas

#section:script_messages
----------messages:(1/24)----------
Executing test class...

#section:testExecute
----------messages:(1/185)----------
command:com.sun.tck.cldc.javatest.CldcTCKCommand
api/javax_microedition/lcdui/Canvas/index.html#GetGameAction
javasoft.sqe.tests.api.javax.microedition.lcdui.Canvas.GetGameActi
   onTests
----------ref:(2/50)----------

Canvas0001:Passed. OKAY
Canvas0002:Passed. OKAY
----------log:(0/0)----------
result:Passed. tests:2; passed:2

test result:Passed. tests:2; passed:2
```

APPENDIX D

Frequently Asked Questions

<These example FAQs are based on the assumption that you are using JavaTest harness version 3.x>

This appendix groups frequently asked question by the following topics:

Configuration

<The following two optional questions are for situations in which there was a previous version of your TCK that used JavaTest harness version 2.x>

{Q: I don't see `.jte` or `.jtp` files anymore, so how do I configure tests?}

{A: When using JavaTest harness 3.x, test configuration is done through the JavaTest harness Configuration Editor. The Configuration Editor prompts you to enter the necessary parameters such as test environment, initial tests, exclude list, keywords, previous tests to run, number of tests in bundle (concurrency), time factor and so on. A test configuration that you create by answering these questions can be saved as a *name*`.jti` file. When you execute the JavaTest harness software from the command line, you can specify loading a `.jti` file in the same way that `.jtp` files were loaded when using JavaTest harness 2.x.}

{Q: Is there an equivalent to the JavaTest harness 2.x `.jtp` file?}

{A: JavaTest harness 3.x replaces the JavaTest harness 2.x `.jtp` and `.jte` files with a `.jti` file. A `.jti` file can be loaded with the JavaTest harness from the command line like a `.jtp` file. You create and modify `.jti` files with the JavaTest harness Configuration Editor or the `editJTI` utility. }

Q: Since `.JTI` files are checksummed, how can I use build scripts to dynamically configure configuration settings for (automated) batch runs?

A: There are two ways to do this. The `editJTI` utility allows automated “offline” modification of the interview file. The batch command allows you to modify the interview answers when invoking the harness in batch mode.

Q: How can we avoid having to re-answer all the interview questions every day?

A: You can save the interview in a `.jti` file and then reload that file when doing the interview for the next build.

Q: Multiple people use similar settings for running the TCK. Do we all need to answer the interview individually and keep track of our own `.jti` files?

You can setup a central repository of `.jti` files, from which people can “prime” their interview. (The files in this repository should be made read-only so that each user does not alter the master copy.) Each user can load the shared master copy, make alterations for their particular task and then save the interview in a new file. This saves everyone from having to answer all the common questions.

JavaTest Harness

Q: Is there a JavaTest harness User's Guide available?

A: Online help is available for the JavaTest harness. You can select the “?” icon or Help->User's Guide from the JavaTest harness menu. There is also a PDF format version of the online help at *TCK_DIRECTORY*`/doc/javatest/javatest.pdf` that can be viewed on-screen or printed out.

Q: How do you move a set of reports to a new location without breaking the links?

A: The JavaTest harness 3.x provides the `editlinks` utility program to reconnect the links after moving the reports.

Q: What factors affect how fast JavaTest runs?

There are many factors, but the key ones are:

- Size of test suite.
- GUI mode vs. batch mode. The GUI has more overhead, so it will be slower.
- Size of work directory—empty or full.
- Using, or not using, a binary test finder (see the top of the `harness.trace` file.

<The following two optional questions are for situations in which there was a previous version of your TCK that used JavaTest harness version 2.x>

{Q: JavaTest does not automatically generate reports anymore, why?}

{A: Reports are now generated by request only. You can create them using the Report menu in the GUI or by using the `-writeReport` option in batch mode.}

{Q: Where is the time-remaining and memory-usage information?}

{A: The Progress Monitor window provides this information. You can open this window by selecting the "magnifying glass" icon in the lower right hand corner, or by selecting Run Tests->Progress Monitor from the JavaTest harness menu.}

Testing an Implementation

Q: Where do I start to debug a test failure?

A: From the JavaTest GUI, you can view recently run tests using the Test Results Summary (click the fourth icon), by selecting the red Failed tab or the blue Error tab. You can also take a look at the raw *testname*`.jtr` file located in the Work Directory. The `.jtr` file lists all the environment variables and echoes the evaluated command strings. Taking these evaluated strings and running them outside of JavaTest harness, on the command line, can often help to debug problems in your environment. See Chapter 9, "Debugging Test Problems," for more information.

Q: How do I restart a crashed test run?

A: Refer to the Test Tree and the *JavaTest User's Guide* for information. The *work_directory*`/jtdata/harness.trace` file may provide additional information.

Q: What results should I expect when I execute the tests within the TCK?

A: When the TCK tests are run with the exclude file (`.jtx`), all tests should pass. This means that from the JavaTest GUI, the Test Results Summary window will show no tests listed on the Failed tab. Remember, to examine all test results, particularly if you have run the [NAME TCK] in separate pieces, you will need to run a final report so that all result files (`.jtr`) are examined. Simply select `jck-report` for your environment selection.

Q: Why are there so many tests in the Exclude List?

A: The JavaTest exclude file (`*.jtx`) contains all tests that are not required to be run. The following is a list of reasons for a test to be included in the Exclude List:

- An error in a reference implementation that does not allow the test to execute properly has been discovered.

- An error in the specification that was used as the basis of the test has been discovered.
- An error in the test has been discovered.

Q: What to do if tests error status “Interrupted (timed out?)”

A: Try reducing the number of “Tests In Bundle” (concurrency) if it was set higher than 1 or increase the “Time Factor”.

Q: The RI will not run, and returns the error: “`Error installing suite: storageOpen(): Permission denied`”

A: Make sure that the directory where the RI is installed has Write permissions enabled.

APPENDIX **E**

Release Notes

<NOTE TO FRAMEMAKER USERS— Because Release Notes are separate from the user guide, this Release Notes appendix should not be used as part of the User Guide book template. Before using the template's FrameMaker book file to create your user guide you must remove this appendix file (a-relnote.fm) from the book file.>

<Release Notes are normally provided in plain-text (ASCII) or HTML files (or both), that are separate from the User Guide. This example Release Notes template, in both ASCII and HTML, is provided here for your convenience to use as you wish.>

Release Notes Template—ASCII

```
Release Notes
[NAME TCK]
Version [VersionNumber]
<Date>

====================
 Table of Contents
====================
1. Introduction
     i. Product Description
    ii. Specifications
   iii. End of Support Notice
2. New Features
3. System Requirements
4. Release Contents
5. Installation
     i. Installation Instructions
    ii. Usage Notes
6. Accessibility Features
7. Known Bugs and Issues
```

```
     i. Installation Bugs
    ii. Documentation Issues
   iii. Software Limitations
    iv. Software Bugs

=================
 1. Introduction
=================

  --------------------
   Product Description
  --------------------
```

<Enter the who, what, why, etc. here.>

```
  ----------------
   Specifications
  ----------------
```

<Enter any specification information here.>

```
=================
 2. New Features
=================
```

<List features new in this version of the software:>

<- this is a description of a new feature>

<- this is another description of a new feature>

```
========================
 3. System Requirements
========================
```

<If the user needs to be aware of platform or memory restrictions, or needs to install additional software packages, use this section.>

```
====================
 4. Release Contents
====================
```

<Describe any key sections of the release. Consider listing the top-level directory structure with a description of each subdirectory, for example:>

<Unzip the distribution into any directory of your choice. It creates the directory <distdir> with the following subdirectories:>

<- subdir1this directory contains this>

<- subdir2this directory contains that>
<- subdir3this directory contains something else>

<Further details about the contents can be described here.>

```
==============================
 5. Installation Instructions
==============================

  ---------------------------
   Installation Instructions
  ---------------------------
```

<Insert any key installation instructions here, and refer the user to the installation / user's guides if appropriate.>

```
  -------------
   Usage Notes
  -------------
```

<If a piece of information is important and needs to be mentioned in order for the user to get the application up and running, use this space.>

```
==========================
 6. Accessibility Features
==========================
```

<If your product is accessible, explain its accessibility features.>

```
==========================
7. Known Bugs and Issues
==========================

  -------------------
   Installation Bugs
  -------------------
```

<Insert descriptions of any installation bugs here.>

```
  ----------------------
   Documentation Issues
  ----------------------
```

<If there are any errors in documentation, detail those here.>

```
  ----------------------
   Software Limitations
```

```
----------------------
```

<If your software has any limitations the user should be aware of, enter those here.>

```
--------------------------
 Software Bugs and Issues
--------------------------
```

<If your project provides bug numbers, you can use a table like this:>

```
 <BUG ID DESCRIPTION>
 <------------------------------------------------------------>
 <123456 Here is bug #123456. It is a short bug.>

<123457 Bug #123457 is next. This is a particularly complicated bug
   which requires a great deal of space in order to explain
   properly.>

<This bug even requires a second paragraph. Talk
about complicated!>
```

<If your project does not use bug numbers, you can use a list like this:>

```
<123456 - Here is bug 123456. It is a short bug.>

<123457- Bug 123457 is next. It is a particularly complicated bug
   which requires a great deal of space in order to explain
   properly.>

<This bug even requires a second paragraph. Talk about
   complicated!>
```

Release Notes Template—HTML

<Some developers, such as Sun, use standardized company templates for HTML-version Release Notes. If you do not already have a standard template, you can use the sample below as an example starting place.>

<The sample template below includes some Section-508 compliant coding. Use of this template should not be considered as complete 508-compliance. You need to check your finished HTML Release Notes against the 508-compliance standards that company requires.>

```
<NOTE--In the example below, material you need to enter is set off
with [square brackets] rather than the <angle brackets> used elsewhere
in this template. This is because HTML uses <angle brackets> as tag
delimiters.
            <html>
            <head>
            <title>Release Notes - [Name of TCK] - [Version X.x]
               </title>
<If you are not using a stylesheet, delete the line below, and all of
the class=[class_name] tags. If you are using a stylesheet, substitute
the appropriate class names where you see [class_name].>
            <link rel="stylesheet" href="[name].css" type="text/css">
            </head>

            <body>
            <table width=100% cellspacing=0 cellpadding=0 border=0
                   summary="This table is for formatting purposes only.">
              <tr>
                <td class=[class_name]> </td>
              </tr>
              <tr>
                <td class=[class_name]>
                  <h1>Release Notes</h1>
                  <h2>[Name of TCK]<br>[Version X.x]</h2>
                  <h4>[Date]</h4>
                </td>
              </tr>
              <tr>
                <td class=[class_name]> </td>
              </tr>
            </table>
            <br>
            <p class=[class_name]>[<a href="#_maincontent">Skip TOC</a>]</p>

            <h2>Table of Contents</h2>
            <dl>
              <dt><a href="#introduction">Introduction</a></dt>
              <dl>
                <dt><a href="#product_description">Product Description</a></dt>
```

```
    <dt><a href="#specifications">Specifications</a></dt>
    <dt><a href="#end_of_support_notices">End of Support Notice</
    a></dt>
  </dl>
 <dt><a href="#new_features">New Features and Improvements</a></dt>
  <dt><a href="#system_requirements">System Requirements</a></dt>
  <dt><a href="#release_contents">Release Contents</a></dt>
  <dt><a href="#installation">Installation</a></dt>
  <dl>
    <dt><a href="#installation_instructions">Installation
    Instructions</a></dt>
    <dt><a href="#usage_notes">Usage Notes</a></dt>
  </dl>
  <dt><a href="#accessibility">Accessibility Features</a></dt>
  <dt><a href="#known_bugs_and_issues">Known Bugs and Issues</a></
    dt>
  <dl>
    <dt><a href="#installation_bugs">Installation Bugs</a></dt>
    <dt><a href="#documentation_issues">Documentation Issues</a></
    dt>
    <dt><a href="#software_limitations">Software Limitations</a></
    dt>
    <dt><a href="#software_bugs">Software Bugs</a></dt>
  </dl>
</dl>


<a name="introduction"></a><a name="_maincontent"></
   a><h2>Introduction</h2>
<p>This section covers the topics:</p>
  <ul>
    <li><a href="#product_description">Product Description</a></li>
    <li><a href="#specifications">Specifications</a></li>
    <li><a href="#end_of_support_notices">End of Support Notice</
    a></li>
  </ul>
```

<If you are not using the “Specifications” and “End of Support Notice” subheads, omit the “Product Description” head below and the section's jump list above, then describe the product directly under the main “Introduction’” heading.>

```
<a name="product_description"></a><h3>Product Description</h3>
```

[Enter the who, what, why, etc. here.]
<h3>Specifications</h3>
[Enter any specification information here.]
<h3>End of Support Notice</h3>
[If support for a release is being stopped, use this section.]
[Top]

```
<The “New Features” section for new releases of existing TCKs. If this
is the first release of a new TCK, this section is not needed.>
            <a name="new_features"></a><h2>New Features and Improvements</h2>
            <p>[List features new in this version of the software:]</p>
            <ul>
              <li>[This is an example description of a new feature.] <br>  </li>
              <li>[This is another example description of a new feature] <br> 
                </li>
            </ul>
            <p class=[class_name]>[<a href="#_top">Top</a>]</p>
            <a name="system_requirements"></a><h2>System Requirements</h2>
<If the user needs to be aware of platform or memory restrictions, or
needs to install additional software packages, use this section.>
            <p class=[class_name]>[<a href="#_top">Top</a>]</p>

            <a name="release_contents"></a><h2>Release Contents</h2>
<Describe any key sections of the release. Consider listing the top-
level directory structure with a description of each subdirectory, for
example>
            <p>Unzip the distribution into any directory of your choice. It
            creates the directory <code>[distdir]</code> with the following
            subdirectories:</p>
            <ul>
              <li><tt>[subdir1]</tt> - [Contains this.]</li>
              <li><tt>[subdir2]</tt> - [Contains that.]</li>
              <li><tt>[subdir3]</tt> - [Contains something else.]</li>
            </ul>
            <p>[Further details about the contents can be described here.]</p>
            <p class=[class_name]>[<a href="#_top">Top</a>]</p>

            <a name="installation"></a><h2>Installation</h2>
            <p>This section covers the topics:</p>
              <ul>
                <li><a href="#installation_instructions">Installation
                Instructions</a></li>
                <li><a href="#usage_notes">Usage Notes</a></li>
              </ul>

            <a name="installation_instructions"></a><h3>Installation
```

```
Instructions</h3>
```

<Insert any key installation instructions here, and refer the user to the product documentation if appropriate.>

```
<a name="usage_notes"></a><h3>Usage Notes</h3>
```

<If a piece of information is important and needs to be mentioned for the user to get the application up and running, use this space.>

```
<p class=[class_name]>[<a href="#_top">Top</a>]</p>

<a name="accessibility"></a><h2>Accessibility Features</h2>
```

<If your TCK Section 508 accessibility features, explain those features (such as keyboard shortcuts) here.>

```
<p class=[class_name]>[<a href="#_top">Top</a>]</p>

<a name="known_bugs_and_issues"></a><h2>Known Bugs and Issues</h2>
<p>This section covers the topics:</p>
  <ul>
    <li><a href="#installation_bugs">Installation Bugs</a></li>
    <li><a href="#documentation_issues">Documentation Issues</a></
   li>
    <li><a href="#software_limitations">Software Limitations</a></
   li>
    <li><a href="#software_bugs">Software Bugs</a></li>
  </ul>
```

<If you only use the Software Bugs subheading, change the main heading to “Known Bugs,” omitting all subheads and the section's jump list.>

```
<a name="installation_bugs"></a><h3>Installation Bugs</h3>
<p>[Describe any installation bugs here.]</p>

<a name="documentation_issues"></a><h3>Documentation Issues</h3>
<p>[Describe any errors in documentation here.]</p>

<a name="software_limitations"></a><h3>Software Limitations</h3>
<p>[Describe any software limitations the user should be aware of
here.]</p>

<a name="software_bugs"></a><h3>Software Bugs</h3>
```

<If your project provides bug numbers, use a table like the

```
following:>
            <table>
              <tr>
                <th class=[class_name] scope="col">BUG ID</th>
                <th class=[class_name] scope="col">DESCRIPTION</th>
              </tr>
              <tr>
                <td>[123456]</td>
                <td>[This is a description of the bug. If it happens to take up more than one
                line, <br> this is what it will look like.]</td>
              </tr>
              <tr>
                <td>[123457]</td>
                <td>[Here is bug #123457. It is a short bug.]</td>
              </tr>
              <tr>
                <td>[123458]</td>
                <td>[This is a particularly complicated bug which requires a great deal of
                space to explain properly.]
                      <p>[This bug even requires a second paragraph. Talk about
                complicated!]</p>
                 </td>
              </tr>
            </table>


<If your project does not provide bug numbers, use a list like the
following:>
            <ul>
              <li>[This is a description of a bug. If it happens to take up more than one line,
                this is what it will look like.] <br>  </li>
              <li>[This bug is next. It is a particularly complicated bug that requires a great
                deal of space in order to explain properly.]
                   <p>[This bug even requires a second paragraph. Talk about
                complicated!] </p></li>
            </ul>
            <p class=[class_name]>[<a href="#_top">Top</a>]</p>


            <hr class="[class_name]">
<End the Release Notes with the appropriate copyright notice.>
```