TCK Project Planning and Development Guide

Version 1.2

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

August, 2003

Please
Recycle

Adobe PostScript

Contents

Preface

TCK Project Planning and Development Guide explains what you need to do to develop a Technology Compatibility Kit (TCK) under the Java Community Process[SM] program. It provides a starting point for a Specification Lead (Spec Lead) and Expert Group and provides an overview of the process involved in creating a TCK. It is intended to address the following questions:

- What is conformance testing?
- What is a TCK?
- Why is a TCK important?
- What components are required for a TCK?
- How do you create a TCK?
- What resources are required to create a TCK?
- How long does it take to develop a TCK?
- How do you plan a TCK development project?

This guide is written for:

- **Technology Specification Leads** (Spec Lead) who scope, plan, and manage the development of a Java™ Specification Request (JSR) including the TCK that is used to test implementations of the that specification.
- **TCK Architects** who design a TCK and its component test suites. Architects are responsible for designing the test suite's overall architecture, the execution and application models, and the functional tests required. The architect is also responsible for designing or configuring a test harness that implements the test execution model.
- **Project managers** who plan and oversee the actual work of TCK development.
- **TCK developers** who create the TCK tests, test suites, and test framework.

How This Book Is Organized

Chapter 1, "Developing a TCK—Introduction," provides an overview of Java Technology Compatibility Kit (TCK) development.

Chapter 2, “Components of a TCK,” describes the components of a TCK as defined by the JCP process.

Chapter 3, “Using Java CTT to Build a TCK,” describes how Sun’s Java Compatibility Test Tools (Java CTT) can be used to help develop a TCK.

Chapter 4, “Planning a TCK,” describes the TCK planning process.

Appendix A, “TCK Appeals Process,” describes the TCK appeals process.

Appendix B, “TCK Project Planning Questions,” describes the questions that various TCK planning documents need to address.

Appendix C, “Measuring TCK Test Coverage,” describes metrics that can be used for measuring TCK Test coverage.

“Java TCK and CTT Glossary,” provides a list of Java CTT-related terms and their definitions.

Related Documents

The following related documents are referred to in this guide and can be found at: `http://jcp.org/en/resources/tdk`.

- ***Java Technology Test Suite Development Guide***: Describes the theory of how to design and write the tests that go into any TCK regardless of which test harness is used. It also provides “how-to” instructions describing how to build your TCK and write the tests that go into your TCK’s test suite. It includes examples of testable assertions and the applicable test code.
- ***JavaTest™ Architect’s Guide:*** Describes the JavaTest™ harness and how to integrate your tests and the JavaTest harness into your TCK.
- ***TCK Tools & Info:*** Describes a Technology Compatibility Kit (TCK) and the Java Compatibility Test Tools (Java CTT) that can be used to help develop a TCK. You can download the Java CTT tools and documentation from this web site.
- ***JCP: Process Document*** (found at `http://jcp.org`): Describes the formal Java Community Process procedures for developing a new Java technology specification or revising an existing specification.
- ***JavaTest User’s Guide*****:** Describes how to use the JavaTest harness. (Note that if your TCK uses the JavaTest harness you should include this guide as one of your TCK deliverables.)
- ***Spec Trac Tool User’s Guide*****:** Describes how to install and use the Spec Trac Tool.
- ***Java API Coverage Tool User’s Guide***. Describes how to install and use the Java API Coverage Tool.
- ***Signature Test Tool User’s Guide*****:** Describes how to install and use the Signature Test Tool.

- **TCK Project Plan Template:** Provides a template in RTF and HTML formats for creating a TCK project plan based on IEEE standards and this *TCK Project Planning and Development Guide*.
- **Java Technology Compatibility Kit User's Guide Template:** Provides an outline and template that can be used as the basis of creating a user's guide for a TCK. The template can be edited to provide the basis of the user manual that tells your customers how to use the TCK you are developing. The template is available in Adobe FrameMaker, Adobe Acrobat, and ASCII plain-text formats.

Accessing Sun Documentation Online

You can find Sun online documentation at the following web sites:

- The Java web site provides general information about the Java 2 platform and related technologies: `http://java.sun.com`.
- The Java Developer Connection℠ web site enables you to access Java™ platform technical documentation on the Web: `http://developer.java.sun.com/developer/infodocs`.
- The Java Community Process (JCP) web site provides information about Java Technology Specifications, JSRs, and the process of developing them: `http://jcp.org`.
- The Sun Product Documentation web site provides online versions of Sun manuals and guides: `http://docs.sun.com`.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at: `docs@java.sun.com`.

CHAPTER 1

Developing a TCK—Introduction

This chapter provides an overview of Java Technology Compatibility Kit (TCK) development and contains the following sections:

Note – Up to date information on TCK development, Java Community Process (JCP) procedures, and the Java™ Compatibility Test Tools (Java CTT) tools are provided on the JCP web site (`http://jcp.org`). Should there be differences between the information provided in this manual and information on the JCP web site, the web site should be taken as definitive.

Scope of the Task

Designing, creating, and testing a TCK is a large task.

To build a TCK you have to:

- Thoroughly understand the technology specification.
- Derive testable assertions from the specification.
- Determine the tests implied by each assertion, and then prioritize them in order of importance for testing.
- Design, develop, and test the tests for each assertion.
- Create a test framework to run, evaluate, and report on tests.

- Assemble the tests, test framework, and documentation into a product suitable for delivery to customers.

In general terms, the task of developing a TCK is as big as developing and testing the specification's Reference Implementation (RI).

Because your TCK is a product that you deliver to your customers, it requires more development work and support than a simple software test suite for internal use.

TCKs and Conformance Testing

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

A TCK is a suite of tests, tools, test framework, and documentation that allows an implementor of an associated Java technology specification to determine if the implementation complies with the specification. This requires that all tests in the TCK are implementation-independent, and based only on the written specifications for a particular Java technology.

For a given platform to be considered "*conformant*" (that is, that it complies with the specification), all of the required TCK tests must pass and meet the testing requirements. Required tests are those tests listed in the TCK documentation as the tests that must be passed. (The RI must also pass all required tests and meet the testing requirements.)

Conformance test development for each feature relies on the existence of a complete specification and RI for that feature. A TCK performs the tests that ensure that all implementations of a specification can meet the *complete* criteria defined by the specification.

To accomplish its purpose, the TCK test must adequately cover the entire specification. When this testing is carried out through a formal process, application developers can be confident that an application will run in a consistent manner across all tested implementations of the same Java technology specification. Without the assurance of knowing that the TCK provides adequate coverage, software developers cannot completely rely upon the published specification because there is no means to trust that any particular vendor's implementation has been correctly constructed.

Conformance testing is important to different groups involved with Java technologies for different reasons:

- Conformance testing ensures that the Java technology does not become fragmented as it is ported to different operating systems and hardware environments or re-implemented by different vendors on the same environment.

- Conformance testing benefits Java technology implementors by ensuring a level playing field for all implementations of that technology.
- Conformance testing benefits application developers using the Java programming language, allowing them to write applications once and then to deploy them across heterogeneous computing environments without modification.
- Conformance testing benefits application developers who wish to incorporate features of a Java technology into their applications by allowing them to write to the Java technology specification rather than to a particular vendor's implementation.
- Conformance testing allows application users to obtain applications from disparate sources and deploy them in any user environment with confidence.

For these reasons, developing a TCK that provides adequate test coverage is a critical task for a JCP Expert Group to undertake when completing a new Java technology specification.

Deciding what level of test coverage, how the test suite will be automated, and the conformance requirements (rules) necessary for passing the test suite are all critical decisions that should be made early in the TCK development process.

When creating a new TCK the Expert Group is not limited in the choice of tools, test suite format, or management of the changes due to appeals. The Expert Group should make choices that best meet the requirements and goals of the technology.

See the *Java Technology Test Suite Development Guide* and *JavaTest Architect's Guide* for detailed information on how to develop TCK tests and test frameworks.

See Chapter 2, "Components of a TCK," for a detailed description of the elements that make up a TCK.

Key Decision Points

When first considering TCK development, there are two key decision points that will have an impact on the amount of resources required, the schedule, and the overall quality of the finished TCK:

1. What is the average amount of desired test coverage? (See "Initial Decision Points" on page 36 and Appendix C, "Measuring TCK Test Coverage," for more information.)

2. How will the test framework be designed, and particularly which test harness will be used to run the tests? Sun provides the JavaTest harness, which is designed for, and used, by many TCKs, but you can use whatever test harness best suits your needs.

Note – You can also use the other optional Java Compatibility Tools (Java CTT) that Sun provides. These tools are designed to ease and speed the TCK development process. They are described in "Sun's Java Compatibility Test Tools" on page 9."

See Chapter 3, "Using Java CTT to Build a TCK," for a step-by-step description of the TCK creation process.

See Chapter 4, "Planning a TCK," for detailed information on planning a TCK development project and estimating the time and resources required.

The Java Community Process (JCP)

The Java Community Process (JCP) is responsible for the development of Java technology. As an open, inclusive, membership organization it guides the development and approval of Java technical specifications, TCKs, and RIs.

In order to ensure both fairness and stability across the Java technology, the JCP has established a formal procedure for developing new technology specifications and revising old ones. This process is known as the "JCP process." The details of the JCP process are described in full on the JCP web site at: `http://jcp.org`.

Because your specification has to be developed using the JCP procedures, you may find it advantageous to assign someone the role of being your "JCP program manager." This person to be responsible for ensuring that your specification, TCK, and RI development meets the JCP requirements in a timely manner.

Java Technology—Components

Under the JCP process, release of a new (or revised) technology specification must contain three primary components:

1. **Specification**. A written specification of technology. There are different kinds of specifications, for example:
   - Platform editions
   - Profiles
   - Optional packages
2. **Reference Implementation (RI)**. The prototype or *proof of concept* implementation of the specification. The RI is required to pass the TCK.

3. **Technology Compatibility Kit (TCK)**. A test kit that Java technology implementors can use to ensure that their work is conformant with the technology specification. The TCK must test all aspects of a specification that impact how conformant an implementation of that specification would be, such as the public API and all elements of the specification. A vendor's implementation of a specification is only considered conformant if the implementation passes the TCK.

As illustrated in FIGURE 1, the development of these three components is best done together in an interactive manner (as opposed to sequentially).

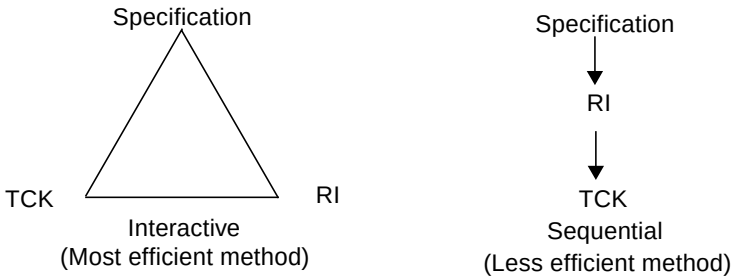

FIGURE 1 Interactive versus Sequential Development

An interactive approach is more efficient because as the RI is developed to implement the specification, what is learned may suggest changes that have to be made to the specification. Similarly, as the TCK is developed and tested against the RI, problems in the RI or in the specification can be revealed. The interactive (triangle) process uncovers loopholes and problems earlier in the process of developing a new specification which in turn results in a more complete and useful specification. Therefore, the *most efficient process is to develop and test all three components together* rather than sequentially one after the other. Note, however, that you should have a stable draft of the specification before you begin developing the TCK and RI.

Conformance Testing vs. Product Testing

Traditional product testing is primarily concerned with:

- Product quality
- Product robustness
- Product performance
- Product functionality
- Product ease-of-use

But those traditional product testing issues are not the concern or focus of conformance testing. The concern of conformance testing is to ensure that an implementation conforms to the specification—regardless of how well it performs.

Therefore, while there are many similarities and overlaps, conformance testing differs from traditional product testing in the following ways:

- **Specification conformance**. The purpose of Java conformance testing is to determine whether an implementation of a technology is compliant with the specification, or multiple specifications, of that technology or technologies. It does not test for performance or ease-of-use.
- **Consistency**. Conformance testing is a means of ensuring correctness, completeness, and consistency across all implementations of a technology specification. In other words, a primary goal of conformance testing is to provide the assurance that an application will run in a consistent manner across all tested implementations of a technology.
- **Scope**. The scope of conformance testing is normally to test those features and areas of an implementation that are likely to differ across implementations. For example, additional effort may be required to test features that:
  - Rely on hardware or operating system-specific behavior.
  - Are difficult to port.
  - Mask or abstract hardware or operating system behavior.
- **Assertion based**. Each TCK test is based on a specific assertion present in a technology specification. The test proves the validity of the implementation being tested.

Similarly, there are significant differences between a TCK that you develop as part of a Java technology specification and a testing system that a software developer creates to test the Reference Implementation (RI) of a Java technology specification (or, for that matter, an end-user application) before releasing it.

For example, some of the differences between a TCK and an internal RI QA test system include:

- **Product**. A TCK is an actual product delivered to customers. An application test-suite, on the other hand, is an internal tool used by a developer to test the developer's products and is never released to the outside world. When QA engineers encounter problems with an internal software test system they can go down the hall and ask for help from the programmer who wrote it, but a TCK is used by others *outside* of your company to test their implementations. This difference has significant implications:
  - Because it is released to the external world, a TCK has to be put through its own cycle of Quality Assurance testing the same as any other product.
  - A TCK must be written and designed for outside end-users. So a TCK has to have a viable user-interface and comprehensible messages. Error-messages have to be understandable and useful.
  - A TCK has to have a good user manual.
  - You need to provide support services as you would with other kinds of products delivered to customers.

- **Open ended**. A TCK has to be designed to test an open-ended set of implementations.
- **Black box**. A TCK performs “black box” testing that does not require knowledge of the internal workings of the software being tested.
- **Configurable**. A TCK has to be configurable for all possible test scenarios.
- **Fairness**. A TCK is intended to provide a fair and equal playing field for all companies developing products under the specification. That is, they all use the same testing materials (the TCK). This requires that you:
  - Provide adequate technical support for the TCK.
  - Provide an appeals process for timely responses to test challenges by users of the TCK.
- **Auditable**. A TCK has to produce auditable results.

Since TCKs measure a Java technology implementation's conformance to a specification, it is normally inappropriate to test unspecified product qualities such as:

- Runtime performance
- Stress / loading capabilities
- Robustness / stability over time
- Other product qualities that are not defined by the specification

These product tests are important to the commercial viability of an implementation and should be tested as part of product quality assurance (QA) testing, however, they are not relevant to conformance testing.

TCK Development—Timeline

There are four basic stages in the JCP process. TABLE 1 summarizes how TCK development fits into these JCP stages.

TABLE 1 Recommended TCK Development Phases and JCP Process Steps

TCK Development Phases	JCP Process
	JSR Initiation Stage
"Preparation Phase" on page 26 • Review materials • Mark up specification first draft • Create TCK project plan • Build TCK development team • Draft legal documents	Community Draft Stage
"Building the TCK Phase" on page 28 • Train TCK development team • Update specification mark-up • Write the tests • Begin work on TCK documentation • Create the test framework • Run TCK against preliminary RI	Public Draft Stage (Public Review cycle)
"Completion Phase" on page 30 • Update specification mark-up • Complete the TCK • Complete the TCK documentation • Run TCK against final RI	Public Draft Stage (final approval cycle)
"Maintenance Phase" on page 32 • Appeals are processed • Exclude list is kept up to date • TCK bugs are corrected	Release and Maintenance Stage

The detailed, specific technology development requirements and procedures are described in the *JCP: Process Document* on the JCP web site at: `http://jcp.org`.

Note – The JCP web site, and particularly the *JCP: Process Document* provides the official, up to date, statement of JCP procedures and requirements. Should there be differences between the information provided in this manual and information on the JCP web site, the web site should be taken as definitive.

Sun's Java Compatibility Test Tools

Sun provides the Java Compatibility Test Tools (Java CTT) that you can use to help develop your TCK. While there may be some portions of your test framework that cannot be provided or built using the Java CTT, using some or all of these tools and resources will help you reduce TCK development time and improve the quality of your TCK. These tools also help measure TCK coverage and quality and thus help track the progress of TCK development.

Sun recommends using these tools to build the core elements of your TCK—particularly in cases where you plan to integrate your JSR with an existing technology in the future. Using common tools will significantly lower the cost of such test suite integration.

The purpose of the Java CTT is to simplify conformance test development and make developing tests and running the tests more efficient. This becomes critical when multiple companies contribute tests to a TCK.

Chapter 3, "Using Java CTT to Build a TCK," describes how these tools and resources can be used together to reduce the time and resources it takes to develop a quality TCK.

The Java CTT includes the following:

- **JavaTest harness**. Configures, sequences, and runs test suites. The JavaTest harness is designed to provide flexible and customizable test execution. It includes everything a test architect needs to design and implement tests for Java technology specifications.

  JavaTest Architect's Guide. Describes the JavaTest harness and how to integrate your tests and the JavaTest harness into your TCK.

- **Spec Trac Tool**. Simplifies the process of tracking assertions within a specification as they change over time, allows test developers to quickly identify assertions in the specification that are candidates for required testing, and provides an assertion ID to bind to a test. The tool provides assertion coverage reports and lists of specification assertions that are, and are not, tested by the TCK. The tool also helps track changes in assertions, and thus helps track tests that need to change as a result of specification revisions.

  The Spec Trac Tool is designed to handle specifications that are created with the Javadoc™ software application that is included as part of Sun's J2SE SDK™.

  Spec Trac Tool User's Guide. Describes how to install and use the Spec Trac Tool.

- **Signature Test Tool**. This tool has two parts:
  - **Signature file generator** that identifies all public and protected members in an API and lists them in a signature file that it creates.

- **Signature test** that can be included in your TCK to verify that each member in the signature file exists in the API under test. The signature test is used to ensure that the entire API is present—no more and no less—than that identified by the specification.

  The Signature Test Tool also checks that the set of superclasses and superinterfaces of a given class or interface, the set of exceptions thrown by a given method or constructor, and the presence of the volatile modifier for a given field, match the specification.

  Signature Test Tool User's Guide. Describes how to install and use the Signature Test Tool.

- **Java API Coverage Tool.** Measures the percentage of public API members (that is, the public methods, constructors, and fields) directly referenced by the TCK tests and compares them to the total number of public API members defined by the specification. In other words, it gives the percentage of API members that are "touched" by the TCK tests. In addition, this tool lists the uncovered public API members per class.

  The Java API Coverage Tool is a static code analysis tool that uses a signature file representation of the specification's API as the source for its analysis. Another use of this tool is to identify API members that are not "touched" by the TCK tests. This can be useful when searching for missing coverage within critical functionality areas of a specification.

  Java API Coverage Tool User's Guide. Describes how to install and use the Java API Coverage Tool.

- ***TCK Project Planning and Development Guide*** (this book). Provides an overview and starting point for the Specification Lead and Expert Group members responsible for developing a Technology Compatibility Kit (TCK) under the Java Community Process program. It is a high-level planning guide for the Spec Lead, TCK architect, and project managers.
- **TCK Project Plan Template**. Provides an editable outline and template that can be used as the basis for creating a written TCK Project Plan based on IEEE standards and this *TCK Project Planning and Development Guide*.
- ***Java Technology Test Suite Development Guide***. Describes how to design and write tests for any TCK. It also provides "how-to" instructions describing how to build your TCK and write the tests that become your TCK's test suite. It includes examples of testable and non-testable assertions and the applicable test code.
- **Java Technology Compatibility Kit User's Guide Template.** Provides an editable outline and template that can be used as the basis for creating a user's guide for a TCK. The template can be edited to provide the basis of the user manual that tells your customers how to use the TCK you are developing. The template is available in Adobe FrameMaker, Adobe Acrobat, and ASCII plain-text formats.
- **Sample TCK**. Provides a collection of tests that demonstrate TCK test development techniques, and illustrate particular test development issues. These tests can be used by test developers as models and examples of how to write their own tests.

The Sample TCK is a complete TCK that complements the *Java Technology Test Suite Development Guide*, Spec Trac Tool, and Signature Test Tool. It includes a Java API specification and a RI. It is run using the JavaTest harness.

CHAPTER 2

Components of a TCK

The following sections describe the components of a TCK as defined by the JCP process. Except for the TCK planning documents, all of the other components are delivered to your customers:

- TCK Planning Documents
- Conformance Rules
- Appeals Process
- Exclude List
- Tests and the Test Suite
- Test Framework
- TCK Documentation
- Audit Process (Optional)

TCK Planning Documents

Once the Java technology specification is drafted, the first TCK components to be developed should be the TCK planning documents which are for your internal use. These plans form the basis for developing the subsequent components that are delivered to customers. The following documents are recommended:

- **Project plan**. Provides a comprehensive description of the TCK as a product.
- **Test plan.** Describes what needs to be tested and how that is to be accomplished.
- **Test integration plan**. Describes how individual tests are incorporated into the test suite.
- **Test specifications**. Describe what each test does and the expected results.
- **Documentation Plan**. Describes the documentation that will be written for the TCK.
- **QA Plan**. Describes how the TCK will be tested against the RI and the specification.

See Chapter 4, “Planning a TCK,” for a detailed description of these plan elements and how to create them.

Conformance Rules

Conformance rules define the criteria a Java technology implementation must meet in order to be certified as “conformant” with the technology specification. In other words, they define what it means to “pass” your TCK. These rules must be clearly documented for those who use the TCK.

Conformance rules often complement the test suite by defining additional conformance criteria required by the specification which may be impossible, or impractical, to check with test applications. Rules can also be used to provide exceptions to testing criteria (for example, identifying situations where passing the test suite is not required).

The Specification Lead, and later the Maintenance Lead, are responsible for establishing and maintaining the conformance rules. An example set of conformance rules can be found in the Java Technology Compatibility Kit User’s Guide Template.

Because all implementations of a technology specification have to meet the criteria established by the conformance rules, the rules function as an impartial, “level-playing-field” environment for companies developing competing products. For this reason, the rules must be written with the same kind of care given to a legal contract or piece of legislation. The rules have to be fair, treat everyone equally, be complete, and be comprehensible.

The common practice is to document the conformance rules in the TCK user guide, including a definition of terms used in the rules.

Conformance rules are tailored for each specification.

Some examples of conformance rules are:

- Other than test configuration files, no test source or binary may be modified in any way.
- Every required test must be able to pass on every publicly documented configuration of a technology implementation.
- The functional programmatic behavior of every binary class and interface must correspond to its specification.
- No sub-setting, super-setting, or modification of the public or protected API is allowed except as defined in the specification.

See Chapter 2 of the “Java Technology Compatibility Kit User’s Guide Template” for more information about, and examples of, conformance rules.

Tests and the Test Suite

A *test suite* is the collection of conformance tests included in your TCK. A TCK test suite is designed to verify that an implementation of a Java technology complies with its specification. A TCK might have multiple test suites, each designed to test a different aspect of the implementation.

Tests are the binaries, source, procedures, and any accompanying information that exercise a particular feature, or part of a feature, as described in the associated specification. Examples of TCK tests are:

- Network and I/O API tests
- API signature and member (constructor, field, and method) tests
- Interactive GUI tests

Most tests should be completely automated, but some tests, such as GUI tests, may require some manual intervention by the person running the test. Tests that require some user intervention are known as *interactive tests*, tests that do not require user intervention are known as *automatic tests*.

Ideally, every testable assertion in the technology specification should be tested. (See Appendix C, "Measuring TCK Test Coverage," for more information on test coverage.)

Assertions are statements of required behavior, either positive or negative, that are made within the technology specification. Each assertion should be tested by one or more tests in the test suite. A single test can test more than one assertion, however, using one test to cover multiple assertions might produce a serious "hole" in test coverage if that test is ever excluded because of a problem with one of its parts. Therefore, the recommended practice is to write separate tests for each assertion.

Test Development Requirements

Tests can be automated, or require user interaction.

Most tests return either a Pass or Fail status but other kinds of status may be permitted if allowed by the conformance rules.

If your test harness detects that a test did not run successfully—that is, the test did not produce a pass or fail result—it should report a status of "Error."

The following general guidelines are suggested for test development:

- Tests should be written so that their pass or fail status is never ambiguous and is clearly presented to the test user.
- All tests should implement the same test interface for the test suite under development.

- All tests should restore the state of the system under test when the test is completed.
- In general, there should be no hard-coding of paths, time-out values, or other hardware specific features.
- No test should depend on a previous test result or action.
- Individual tests should normally test a single assertion (or part of a single assertion).
- The test should report adequate and meaningful failure diagnostics.

Ideally, TCK tests are developed to exhibit the following characteristics:

- Hardware and software platform independence (all tests use Java code with native and/or platform specific code when no other means exists)
- Implementation independence with no reliance on any implementation specific API or behavior
- Designed so that the tests clearly determine pass/fail status

If a specification defines optional functionality in its API, then tests should be written to cover that optional functionality if it is implemented. The TCK should be configurable so that the user can identify existing optional behavior and be confident that it is tested by the TCK.

Tests are defined as “required” if they must be passed. A “non-required” test is one that is on the Exclude List, or that tests optional functionality that has not been implemented in the product being tested.

Test Code Conventions

Test source code should always be delivered as part of the TCK product. It is expected that test users will need to review the test source code whenever necessary in the course of using the TCK to debug test failures of their implementations. For this reason it is important to follow general source code writing conventions, such as formatting and well chosen variable names. It is also important that the code be adequately documented and understandable by reasonably knowledgeable users. Source code should be checked to make sure that is does not contain rude, indecent, or vulgar method names, variable names, or other text.

See the *Java Technology Test Suite Development Guide* for detailed information on tests and the test suite.

Appeals Process

The Specification Lead is responsible for establishing a clearly defined TCK appeals process to address challenges to the tests contained in the TCK.

Under the JCP process, technology implementations developed for your technology specification must pass your TCK tests in order to be certified as compliant with that specification. Someone engaged in developing such implementations however may wish to challenge the fairness, validity, accuracy, or relevance of one or more of your TCK's tests. Therefore, the JCP requires you to provide an appeals process by which implementors can challenge TCK tests. If a test is successfully challenged (that is, found to be invalid) through the appeals process, that test is removed from the TCK or modified to correct the problem.

The appeals process is an escalation process managed by the Maintenance Lead, in which challenges to conformance tests by implementors are evaluated and either accepted or rejected. The Maintenance Lead is appointed by the Specification Lead and is responsible for maintaining the technology specification, RI, and TCK.

- The appeals process must provide a common method for implementation developers to challenge one or more tests defined by the Specification's TCK.
- The appeals process must be described in the documentation included with the TCK.

Technology implementors do not have to pass TCK tests that have been successfully challenged. The common practice is to list successfully challenged tests in an Exclude List as described in the following section.

See Appendix A, "TCK Appeals Process" for more information on how to create your appeals process and the requirements that it must meet.

Exclude List

When a test is found to be invalid through the appeals process it should be replaced or removed from the test suite.

In most cases, individual test replacement is expensive and must be coordinated with all TCK licensees. Therefore, tests are typically removed from the test suite until it is practical to update the entire TCK with correct and/or new tests. The method of removing a test from the test suite is generically referred to as using an "*Exclude List*."

An Exclude List itemizes tests that do not have to be run for the specific version of the TCK being used. Implementors are not required to pass—or even run—any test or test on the TCK's Exclude List.

An Exclude List is a practical method of managing test challenges without having to release an entire new test suite whenever an invalid test must be removed. The recommended practice is to use a test harness that automatically identifies tests on the Exclude List and does not run them. The JavaTest harness, for example, provides this functionality.

An Exclude List should be associated with each version of a TCK. It is the responsibility of the Maintenance Lead to provide, and maintain, an up to date Exclude List for the TCK. The initial version of an Exclude List can be bundled with the TCK, or it can be made available separately. A documented method must be provided to TCK users for obtaining the current Exclude List. One way of doing this is to have a web site from which an up to date Exclude List can be downloaded. A method of identifying different Exclude List versions must be provided so that testers can easily make sure that they are using the correct version.

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

- An error or ambiguity 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 test harness or test agent).
- An implementation dependency such as thread scheduling model or file system behavior, is identified.

Note – Typically, TCK users are never permitted to alter or modify Exclude Lists used for certification test runs. For more information about using an Exclude List as part of your appeals process, please see Appendix A, "TCK Appeals Process."

Test Framework

A test framework includes the supporting code and tools that are used to manage the TCK testing process. A test framework will usually be customizable and configurable, to meet the requirements of the TCK for any user environment.

A test framework usually consists of at least the following components:

- A test harness as described in "Test Harness" on page 19
- Shared test libraries (common test code used by multiple tests). For example, code used by several tests to open a specialized network connection using a technology or implementation specific protocol.
- An optional test agent as described in "Test Agent" on page 20
- Testing scripts and other test execution / management command code as needed.

- Test harness configuration capability that allows the user to specify necessary variables (see "Test Framework Configuration" on page 21 for details).

Ideally, a TCK should provide all framework components necessary to run the TCK. Occasionally, however, implementation-specific features of the specification may make creating generic components impossible. In those cases, the test users may be required to create such components themselves. For example:

- Application Management Software (AMS)—which is sometimes referred to as Java Application Manager (JAM)—on the device being tested. This is software that resides on (or is downloaded to) the device so that it can handle and respond to test bundles sent by the test harness.
- Implementation of a communications channel between the device under test and the host system running the test harness so that the two can communicate. For example, some TCKs implement a communications channel using the HTTP protocol.

Implementation-specific test components may vary widely in complexity depending on the test environment and the test harness being used. It is up to your licensees to provide these components as needed using whatever tools or techniques are appropriate for their implementation.

Providing example code for such components as part of your TCK is recommended. For example, providing the test framework components necessary to test the RI gives your licensees a head start in creating their own components because they can examine how your components are used to test the RI.

Note – At the present time, Java CTT does not provide specific framework code for J2ME devices (MIDP devices, for example) or for J2EE deployments. Developing this kind of framework can be complex. Sun hopes to provide additional assistance and improvements in these areas in the future.

Test Harness

A test harness typically provides:

- Test selection of one or more tests to execute.
- Sequencing of tests, allowing them to be loaded and executed automatically.
- Automated reporting capability, to minimize manual errors.

Typically a test harness runs on a test server while managing the testing process on a target device or workstation.

The test harness must be capable of running tests on any potential target device, or if permitted by the conformance rules, the test harness may run tests on a computationally equivalent emulated environment or a breadboard.

A test harness should facilitate the task of making sure that all required tests were run, and all of the required tests are passed for each tested configuration of an implementation.

A test harness needs to have some way that the user can configure it and the rest of the test framework for the test environment (hostnames, IP addresses, file and directory locations, which tests to run, where test results are to be stored, communications parameters, test parameters etc). See "Test Framework Configuration" on page 21 for detailed information.

Most TCK developers at Sun run their tests using the JavaTest harness as described in the following section.

Sun's JavaTest Harness

TCKs developed at Sun use the JavaTest harness for test execution and test suite management. The JavaTest harness is a Java application that runs on most workstations and PCs.

The JavaTest harness is designed to execute tests on varying Java technology implementations from powerful servers to small consumer and embedded systems. The JavaTest harness provides the following functionality:

- Graphic User Interface (GUI) for ease of use
- Highly flexible and configurable
- Robust test framework for running tests
- Graphic and file-based display of test results
- Failure analysis
- Reporting
- Test auditing
- Distributed testing environment support

The JavaTest harness is one of Sun's Java Compatibility Test Tools and is available to Expert Groups developing TCKs. See the JCP web site: `http://jcp.org` for more information.

Test Agent

A test agent is a Java application that receives tests from the test harness, runs them on the software being tested, and reports the results back to the test harness. Such an agent is useful when it is impractical, or impossible, to run the test harness on the same device as the implementation under test due to memory or API constraints.

- If the technology implementation is being tested on the same machine as the test harness, you may, or may not, need to have a separate test agent. In other words, when the technology implementation under test and the test harness are running on the same system, the TCK's test harness may be able to function without an agent. In other cases, the software you are testing may require a test

agent even if it is running on the same machine as the test harness. In such cases you can use a separate test agent running as an independent process on the same machine as the test harness.

- If the technology implementation under test is *not* running on the same machine as the test harness, then you need to install and run a test agent on the system or device being tested. When the agent and test harness are on different systems, they communicate with each other over a network or other connection.
- Occasionally a TCK user will want to test multiple implementations in parallel to reduce test run times. Providing the capability in your TCK to use multiple agents running on separate systems or devices is a good way to provide this functionality.

A general purpose agent called *JavaTest Agent* is included with the JavaTest harness.

If a new application model or device communications channel is defined by the specification or required for TCK test execution, it may be necessary to create a custom agent for your TCK. Writing a custom agent can be a complex and time-consuming task, so such work should be factored into your TCK project plan and schedule.

Test Framework Configuration

In order to run TCK tests, the test framework has to be configured for the test environment. In other words, those who use the TCK to test their implementations have to be able to specify hostnames, IP addresses, file and directory locations, agent usage and configuration, which tests to run, where test results are to be stored, communications parameters, test parameters, and so on, for their environment.

Test suites that run in many different types of environments (for example, on different kinds of hardware and operating system platforms) generally require more configuration information than test suites that run in more rigidly defined environments.

The methods of configuring the test framework will vary from one TCK to another. The JavaTest harness, for example, provides a wizard-like interface, called a *configuration editor*, to interview the user about the environment in which the TCK tests will be run. The configuration editor prompts the user through a series of environment questions called a *configuration interview* to identify needed parameters such as IP addresses. The goal of a configuration interview is to gather information required by the test framework to determine which tests to run and how to run them. Based on the information provided by the user during the interview, the editor then properly configures the TCK.

If you choose to use the JavaTest harness, writing and debugging the interview is an essential part of TCK development. Similarly, if you use some other test harness, you have to provide some method of configuring your TCK framework.

TCK Documentation

TCK documentation includes a number of components as described below.

TCK User's Guide

A TCK should have a TCK user's guide to document the TCK. You may also need to include a separate user's guide for the test harness. At a minimum, a TCK user's guide should contain the following:

- Introduction or overview of the TCK.
- Description of how install and configure the tests and test framework.
- Description of how to run the TCK against the RI to verify configuration.
- Description of how to run and use the TCK to test a technology implementation.
- Conformance rules that the implementation under test must meet.
- Appeals process for handling test challenges by users.

Sun's Java CTT includes the *Java Technology Compatibility Kit User's Guide Template.* You can use this template to create your own TCK user's guide if you wish. It is provided in Adobe FrameMaker, Adobe Acrobat PDF, and ASCII text formats.

Test Documentation

The tests themselves must also be documented as follows:

- **Source code**:
  - Individual test source code
  - Specialized test framework source code such as agent, examples, shared test libraries, and so forth that can assist the user in solving test execution problems and debugging test failures.
- **Test descriptions**. Test descriptions are test meta data used by the test harness to locate and run a specific test. (See the *JavaTest Architect's Guide* for details.)
- **Test specifications**. Test case specifications describe in logical terms what a test does and the expected results. A test case specification refers to one or more assertions that a particular test is to verify, as derived from the related API specification.

Release Notes

Release notes are typically provided in HTML or plain ASCII text format and usually include some or all of the following information:

- Product Description
- Produce specifications
- New Features (if this is an update to an existing TCK)
- System Requirements
- Release Contents
- Installation instructions
- Usage Notes
- Accessibility Features
- Known Bugs and Issues

Audit Process (Optional)

As an option, a formal audit process may be used to better ensure the integrity of a consistent self-certifying conformance testing program.

An audit can be accomplished through:

- Physical testing. This is often impractical and usually quite costly due to differences in hardware and/or operating systems that could be used to support the implementation under test.
- Test results verification.

If a test results verification program is implemented, then an independent third party may be used to verify that all the required tests were run and have passed by examining the test reports generated by the test harness.

The JavaTest harness supports an automated test results verification feature for use in such an audit program.

CHAPTER 3

Using Java CTT to Build a TCK

This chapter describes how to develop a TCK using the Java CTT package of tools, documents, samples, and templates. It contains the following sections:

The steps and task sequence presented in this chapter assume you are using the Java CTT. These steps are recommended by Sun as the most cost-efficient way to develop a TCK. However, so long as you meet the requirements mandated by the JCP, you can develop your TCK however you wish.

Note – The Java CTT can be used to efficiently develop much of the software required for your TCK. However some portions of the test framework such as application management software or communications protocols may be specific to a given technology implementation and/or test environment and outside the capabilities of the supplied Java CTT framework code. Therefore such functionality may have to be developed independently by you using other tools or techniques. Depending on your implementation or test environment, this may require significant development resources.

Creating a TCK—Overview

For ease of discussion, the process of creating a TCK is broken up into four phases keyed to the stages of creating a new JSR described in the *JCP: Process Document*. This recommended TCK development sequence is summarized in TABLE 1 on page 8.

While the sequence and timing of how you create your TCK is up to you, Sun recommends that you follow the timeline presented in this guide. If your TCK development does not proceed in sync with the corresponding JCP steps for specification creation, you risk being caught at the end of the process with a delay in final approval of the specification because your TCK is not ready.

Preparation Phase

Preparation for TCK development usually gets underway after the first draft of the preliminary specification is written, and before it is submitted to the JCP for Community Review.

Sun recommends the following preparation steps:

1. **Download the Java CTT components.**

   Go to the JCP web site (`http://jcp.org/en/resources/tdk`), click through the Java CTT Evaluation License Agreement (or sign the Redistribution License Agreement), and then download the tools.

   - **Java CTT Evaluation License Agreement** allows you to download and use the Java CTT for evaluation purposes.
   - **Java CTT Redistribution License Agreement** allows you to include Java CTT components such as JavaTest harness and Signature Test in the TCK that you deliver to your customers.

2. **Develop the expertise you will need to manage a TCK project.**

   Building a TCK is a complicated endeavor. Thoroughly understanding all aspects and requirements before commencing work will save both time and money in the long run. TCK managers should familiarize themselves with the following:

   - The *JCP: Process Document* (`http://jcp.org`) for the specification development timeline-sequence required by the JCP.
   - This *TCK Project Planning and Development Guide* to understand the overall scope and process of TCK development, and to provide a basis for estimating the time and resources you will need.

   According to their various roles, different members of your TCK development team need to be familiar with different Java CTT documents. FIGURE 2 below illustrates the relationship, and target audiences, of the full Java CTT documentation set. For example, this *TCK Project Planning and Development Guide* is intended for Specification Leads, TCK architects, and program managers; while the *Java Technology Test Suite Development Guide* is written for those who create and test TCK tests.

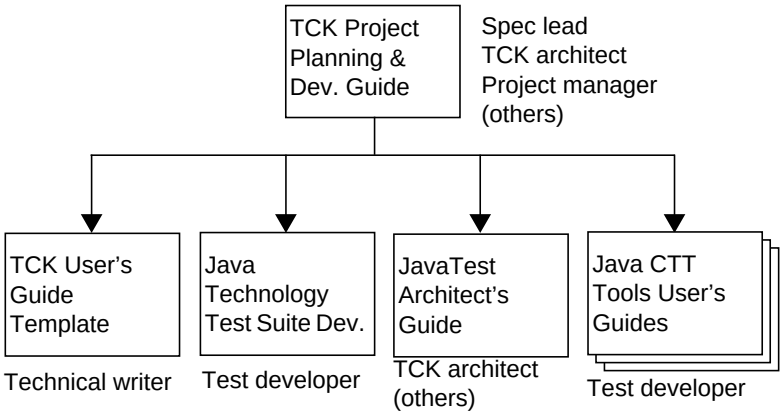

FIGURE 2 Java CTT Documentation Set and Target Audiences

3. **Evaluate the Java CTT package of tools, templates, samples, and documentation.**

   Look over, and try out, the Java CTT. This evaluation will help you decide if it will be of assistance to you in developing your TCK. (If you decide to use the Java CTT you must sign the Java CTT Redistribution License Agreement.)

4. **Mark up the first draft of your specification.**

   Both TCK planning and test creation are based on the testable assertions in your specification. Therefore, your first actual TCK development task is to identify your specification assertions and classify them appropriately. This process is called *specification mark-up*.

   Specification mark-up is an on-going process as the specification goes through revisions to incorporate feedback from the different review processes. Since these revisions change the specification assertions, you need to regularly update the mark-up to ensure that all assertions are correctly classified.

   Sun's Spec Trac Tool is designed for this purpose. The recommended practice is to use Spec Trac Tool for specification mark-up, measuring how adequately your tests cover the specification, and tracking changes in the specification over time.

5. **Create your TCK project plan.**

   Once you have marked up your draft specification, you have an idea of how many testable assertions it contains. That is one of the key factors in estimating the time and resources needed to create your TCK.

   A TCK project plan specifies the TCK coverage goals, overall test architecture and design, resources needed to complete the project, tools to be used, how it will be tested and delivered to customers, the development schedule, personnel assignments, and so forth.

   For detailed information, see:

- Chapter 4, "Planning a TCK," for a detailed description of the planning process and help in estimating the resources you will need for test development.
- Appendix B, "TCK Project Planning Questions," for information on the questions that various TCK planning documents need to address.
- See also the TCK Project Plan Template available at: `http://jcp.org/en/resources/tdk`.

6. **Begin building your TCK development team.**

   Now that you have an estimate of the scope of the task, you can begin forming the TCK development team.

   See "Team Personnel" on page 38 for information on the different roles involved in TCK development and "Resource Estimates" on page 39 as an example of how to estimate time and personnel requirements.

7. **Draft your license terms and appeals process.**

   Before you can submit your draft specification for Community Review under the JCP process, you must draft your proposed licensing terms for both the TCK and the RI, and your proposed TCK test appeals process.

   See the *JCP: Process Document* (`http://jcp.org`) for details.

Building the TCK Phase

TCK development can begin as soon as the first draft of the specification is written. However, as a practical matter, many developers prefer to commence TCK development only after the specification has been revised based on feedback (if any) from the Community Review.

Regardless of when TCK development starts, the common—and recommended—practice is to have the TCK framework development close to complete, and test development well underway, by the time your specification is submitted to the JCP for Public Review. If your TCK development is not far along by that point, release of your specification may be significantly delayed because the JCP Executive Committee will not vote final approval for a specification until the TCK is completed and the RI passes.

Sun recommends the following steps:

1. **Train the TCK development team (if necessary).**

   Developing TCK tests and creating the test framework is highly specialized work. If your developers have not had prior experience in this area, devoting some time to training them at the beginning of the project is a cost-effective way of saving time and money in the long run.

   Suggested training might include having your developers:

- Familiarize themselves with *Java Technology Test Suite Development Guide* and work through the various exercises it contains.
- Familiarize themselves with Part 1 of *JavaTest Architect's Guide*.
- Work through the *JavaTest Architect's Guide* tutorial and the chapter that describes creating a simple test suite.
- Set up and run the Sample TCK that is included in the Java CTT.
- Review test sources, signature test, and coverage files contained in the Sample TCK.

2. **Update the specification mark-up (if necessary) and generate a new assertion list.**

   Since TCK test development is based on the testable assertions in the specification it is essential that you work at all times from a complete and up to date assertion list. If the specification has changed since the last time you classified its assertions, you need to do so again.

   Spec Trac Tool has the capability to track and report specification changes. (See *Spec Trac Tool User's Guide* for details.)

3. **Begin designing your tests and writing the test specifications.**

   Your tests have to be designed to test implementations for conformance with specification assertions.

   See *Java Technology Test Suite Development Guide* for further information.

4. **Include the signature test in your TCK.**

   See *Signature Test Tool User's Guide* for further information.

5. **Begin designing and creating your test framework.**

   Your test framework will need a test harness and possibly one or more additional components:

   - Test harness.
   - JavaTest interview (if you are using JavaTest harness).
   - Test finder or finders.
   - Test agent. (The JavaTest harness includes the Java Test Agent that you can use, or you may need to write your own custom agent.)
   - Test script or scripts.
   - Exclude List (which can be empty).
   - Other framework components as might be needed. (Depending on the specifics of your implementation, additional framework components may require extensive development work.)

6. **Integrate your tests with the test harness.**

   See *JavaTest Architect's Guide* for general information about designing and building test frameworks, and *JavaTest User's Guide* for information about the JavaTest harness.

7. **Begin testing and de-bugging your tests and test framework.**

   If you have a stable Reference Implementation (RI) available, run your tests against the RI to find and correct problems. (Lack of an RI at this stage will delay your overall specification development because your specification cannot win final approval until you have a completed TCK and an RI that passes all of your TCK tests.)

   Run the TCK against the RI to check for, and correct, problems with both the tests and the test framework. Note that a test could fail for one of two reasons. A valid test could fail because there is a problem with the RI (that is, the RI does not correctly implement the specification or meet a conformance requirement). Or, if the RI is correct, the test could fail because of some bug or error in the test itself.

8. **Produce preliminary TCK reports.**

   Once you have a working TCK and RI, you should begin producing the following reports to check and evaluate your progress:

   - Test run results showing that the RI has passed (or failed) the individual tests.
   - Assertion coverage reports generated by Spec Trac Tool.
   - API coverage reports generated by Java API Coverage Tool.

9. **Begin work on the TCK User's Guide**

   See the Java Technology Compatibility Kit User's Guide Template for more information.

10. **Update your TCK Project Plan.**

    Keeping your TCK Project Plan up to date as the work evolves is a useful—and cost-effective—management tool.

Completion Phase

Your TCK (and RI) must be completed before you can submit your final specification to the JCP Executive Committee for final approval ballot. The specification will not be approved unless the TCK can be run against the RI with no test failures.

Sun recommends the following steps:

1. **Update the specification mark-up (if necessary) and generate a new assertion list.**

   If the specification has changed you need to ensure that your TCK development is based on a complete and up to date specification mark-up and assertion list. To do this you should run the Spec Trac Tool's `specdiff` and `track` commands.

2. **Complete your TCK including the test framework.**

   Make sure that:

- All TCK tests can be run against the RI without failure and the RI meets all testing requirements.
- Your test suite meets the coverage criteria targets set forth in your TCK Project Plan. Assertion coverage can be measured by the Spec Trac Tool and method coverage by the Java API Coverage Tool.
- Your test framework includes all the components and example code (if necessary) for your customers to test their implementation in the appropriate environments.
- Your Exclude List is up to date and accurate.
- The redistributables that you will be delivering to your customers are correctly packaged and ready to go.

3. **Put your TCK through a formal QA test program.**

   As with any other software delivered to your customers, your TCK should be rigorously tested in a formal QA program.

4. **Complete your TCK documentation.**

   Make sure that your documentation set includes the following:

   - An adequate and comprehensible TCK User's Guide describing how to install, configure, and use your TCK.
   - The Conformance Rules that your customer's implementations have to meet.
   - An appeals process that customers can use to challenge TCK tests.
   - Test harness documentation, if needed.
   - Release Notes (or Readme file) as needed.

5. **Run your TCK against your RI to produce your final reports.**

   To ensure that your TCK and RI meet the quality standards set forth in your Project Plan, you need to compare the final reports to your plan goals. For example, with the Java CTT, you would generate the following reports:

   - Final test-run reports from the JavaTest harness showing that the RI passed all of the TCK tests.
   - An assertion coverage report generated by the Spec Trac Tool.
   - An API coverage report generated by the Java API Coverage Tool.

6. **Prepare for TCK and RI delivery to your customers.**

   Make ready, and test, whatever methods you will use to deliver your finished TCK and RI to your customers. For example, you might have your customers download your TCK and RI from a web site. (Be sure you have signed the Java CTT Redistribution License Agreement.)

Maintenance Phase

The specification maintenance phase begins after your specification has won final approval from the JCP Executive Committee and it has been posted to the JCP web site as a final release. The maintenance phase includes the following:

- A Maintenance Lead (ML) is designated.
- The ML responds to test challenges (if any) and manages the appeals process as described in your TCK documentation.
- The ML maintains and keeps the Exclude List up to date.
- The ML keeps track of suggested or proposed changes that might be included in some future release of the specification, RI, or TCK.

CHAPTER 4

Planning a TCK

This chapter describes a recommended TCK planning process that will help you scope out and organize the development of your TCK. It contains the following sections:

Developing a TCK is a large and complicated task, but working through the basic planning steps discussed in this chapter can simplify the process and save significant time and human resources during the course of the development cycle. These guidelines will also help you estimate the time and resources needed to complete the project. The procedures contained in this chapter are written for large, complex TCKs. If your TCK is smaller, you may wish to simplify, or omit, some of these steps.

You can use the TCK Project Plan Template which provides an outline, instructions, and some boilerplate that can be used as the starting point for writing TCK plans. The TCK Project Plan Template is available at: `http://jcp.org/en/resources/tdk`.

Initial Questions

The first step in planning a TCK is to consider the initial questions that have to be addressed before you begin planning. You have probably already thought about most or all of these questions, and it is not necessary at this stage to have complete detailed answers to every one of them, but listing them here will make your planning process easier.

- **Does your specification define its own application model?**

  An application model is the method used for downloading an application into a particular device. It must include the definition of a specific Java class or interface, a method of which is called by the runtime environment as the first method to run when the application starts. Some examples are:

  - The default application model described in JVMS
  - Applets in J2SE
  - Midlets in MIDP
  - Xlets in Java TV
  - Java Card Applets in Java Card.

  In regards to planning your TCK, your answer to this question determines whether you can use a pre-existing test framework that supports this application model (such as the JavaTest Agent and test framework code supplied with JavaTest harness), or whether you will have to write your own.

- **How many testable assertions does your specification define?**

  The number of testable assertions is directly related to the number of tests you have to write, and thus can be used to estimate the overall size of your TCK and the time and resources you will need to complete it.

  Sun recommends using Spec Trac Tool to classify and track the testable assertions in your specification.

- **How many API members does your specification define?**

  While not as precise as the assertion count, it is often much easier to estimate the number of API members which for most APIs is also related to the number of tests you will need to write, and thus can be used to estimate the overall size of your TCK and the time and resources you will need.

  Sun recommends using the Java API Coverage Tool to identify, count, and track your API members.

- **Do you have APIs that require functional tests**?

  This question distinguishes between two kinds of tests:

  - *API member tests* (sometimes referred to as *class and method* tests) are designed to verify specific semantics of individual API members. (See *Java Technology Test Suite Development Guide* for more information about class and method testing issues.)
  - *Functional tests* are designed to verify higher level functionality supported by a specification, often as a result of cooperation among multiple API members.

  For TCK planning purposes, the distinction between these two kinds of tests is important because member tests are usually simpler and easier to write than the more complex functional tests, but functional tests may be needed for effective coverage of the specification.

For example, with functional tests:

- The visual behavior of graphical user interface (GUI) components may require interactive testing because the test user has to determine whether the components are visualized correctly. (By comparison, the simple pass-fail status of API member tests can usually be determined automatically by the test itself.)
- Tests for network protocols may require running different components of the tests in different runtime environments on different network hosts.
- Serial form testing for serializable classes have to verify that the serializable object can be correctly serialized and deserialized and verify the serial form stream against the specification.

Functional testing is typically implemented using object libraries (frameworks) that contain the TCK code common to the specific area of functional testing. These libraries have to be written or otherwise obtained.

If your specification contains many APIs that require functional tests, your TCK test suite will be more complex and take longer to write. Libraries may also have to be created to support them. And, of course, your QA process of de-bugging the tests will also take longer.

- **Does your specification have methods/APIs that can return different values depending on different environments, or on different implementations of the specification?**

  For example:

  - `java.util.Locale.getAvailableLocales()` may return different values depending on what locales are supported by this particular implementation.
  - `java.util.Date.toString()` may return different strings depending on the time zone settings of the underlying OS.

  The more implementation-dependent values contained in your specification, the more configurable your TCK has to be to accommodate different environments and implementations, and that makes your tests more complex. This can affect the time required to develop the tests and the TCK as a whole because the design of your TCK has to take into account this configurability.

- **How likely it is that the specification will be revised multiple times?**

  Is this an API for a well-known stable technology or is it an API for a technology that is undergoing extensive changes during TCK development? If the API is evolving rapidly, individual tests may have to be added or updated and the TCK as a whole may have to be reworked in the middle of its development cycle. When planning time lines and schedules, these possibilities need to be taken into account.

  If the specification is undergoing changes, you should consider using the Spec Trac Tool to track and monitor assertion changes to help identify the associated impacted tests.

- **How complex is the data structure represented by the API?**

  For example, is it relatively easy to get an instance of a class/interface to test or do you have to go through a long chain of API calls to obtain an object to test? API complexity affects the length of time it takes to write your TCK tests. The more complex the API, and the more calls required, the longer it will take to write and debug the test.

- **Are there any aspects of application or data delivery that are intentionally not defined by the specification but left to the implementation?**

  For example, CLDC does not require any specific type of connectivity to be present. Nor does Java TV, where the way of delivering broadcast data to the Set Top Box is device-dependent.

  If there are undefined aspects left to the implementation, your TCK may have to be designed to leave implementation of some parts of the test execution framework to TCK users. Not only is this a design and coding issue, it also makes TCK debugging more difficult because you have no control over whether the user-created code functions correctly.

Initial Decision Points

There are several key decisions that need to be made before you can develop your plans:

- **Assertion coverage**. What is the overall amount of assertion test coverage desired for your API? Obviously, the greater the coverage, the more tests you will require, and the more time and resources you will need to develop and debug your TCK. The best practice is to strive for at least 75%assertion breadth coverage.

  See Appendix C, "Measuring TCK Test Coverage," for more information on test coverage.

- **API Coverage**. Assertion-based testing is the most accepted means to check an implementation for conformance. Therefore, TCK completeness is normally defined by the amount of its assertion coverage. API member coverage, however, can provide an additional means for measuring the completeness of a TCK. What amount of member coverage (that is, direct references by tests to public methods, constructors and fields) is desired for the public API's defined by your specification? The recommended practice is to try to provide at least 90% API coverage.

- **Test harness**. Which test harness will you use? You can use the JavaTest harness that Sun provides as part of the Java Compatibility Test Tools (Java CTT), or you can use another test harness.

  (Note that the "Resource Estimates" section of this chapter assumes that you are using the JavaTest harness.)

- **Test agent**. If using the JavaTest harness, will you need a test agent as described in "Test Agent" on page 20? If so, will you be using the JavaTest Agent provided with the JavaTest harness, or will you be writing your own test agent?
- **Java CTT tools**. Will you be using the Java CTT tools that are described in "Sun's Java Compatibility Test Tools" on page 9?"

  (Note that the "Resource Estimates" section of this chapter assumes that you are using the Java CTT.)

TCK Plans

Once your preliminary specification is drafted and your initial decisions made, you need to draft the plans for your TCK. These plans form the basis for developing the subsequent components that are delivered to your customers. Experience has shown that careful TCK planning at the start of the project saves time and money in the long run.

You can use the TCK Project Plan Template which provides an outline, instructions, and some boilerplate that can be used as the starting point for writing TCK plans. The "TCK Project Plan Template" is available at:
`http://jcp.org/en/resources/tdk`.

Conceptually, there are a number of different elements that go into a set of TCK plans:

- **Overall TCK Project Plan**. Provides a comprehensive description of the TCK as a product. (See "TCK Project Plan" on page 47 for some of the questions that the overall project plan should address.)

  Keep in mind that developing and delivering a TCK to customers is more complex than writing testing software to be used internally in-house. See "Conformance Testing vs. Product Testing" on page 5 for more information on this issue.
- **Preparation Plan** (if needed). Describes what preparations (if any) are needed before TCK development work can begin. For example, research, prototyping, developer training, acquisition of hardware or tools, creation of infrastructure, and so on. (See "Preparation Plan" on page 48 for some of the questions that this plan should address.)
- **TCK Design Document**. Describes what needs to be tested and approaches for how testing will be accomplished. This document also describes how the individual tests are incorporated into the test suite, and how the test suite is integrated with the test harness. (See "TCK Design Document" on page 48 for some of the questions that this document should address.)
- **Test Specifications**. Describes what each test does and the expected results. Typically, these test specs are also included with the tests, or as part of the TCK documentation, so that they are available to test users. In other words, this

component is both part of the plan, and part of your TCK documentation. (See "Test Specifications" on page 49 for some of the questions that this document should address.)

- **Documentation Plan**. Describes the documentation that will be written for the TCK. (See "Documentation Plan" on page 50 for some of the questions that this plan should address.)
- **QA Plan**. Describes how the TCK will be tested as a product to be delivered to customers. (See "QA Plan" on page 51 for some of the questions that this plan should address.)

If your planning information is not very complicated or lengthy, some or all of the above elements can be combined into the same document.

See Appendix B, "TCK Project Planning Questions," for a list of questions that these different plans should address.

Team Personnel

A TCK development project contains within it a number of different roles. While one person can perform multiple roles, it is useful for planning purposes to think of the roles separately:

- **Specification Lead**. Heads the Expert Group and is responsible for overseeing all aspects of developing (or revising) a Java technology specification. This includes development of the Reference Implementation (RI) and Technology Compatibility Kit (TCK).
- **Maintenance Lead**. Responsible for maintaining an existing technology specification and related RI and TCK. Manages the TCK appeals process and any revisions needed to the specification, TCK, or RI.
- **Program Manager.** Because your specification has to be developed using the JCP procedures, someone needs to be responsible for ensuring that your specification development meets the JCP requirements in a timely manner.
- **TCK Architect**. Leads the TCK development team and designs the overall test architecture and testing framework. Lists, or oversees the identification of specification assertions that need to be tested and the corresponding test specifications. Determines how the tests and test suite are integrated with the test harness.
- **Test Developer**. Writes the individual tests and implements the testing framework (as needed). Debugs test problems. If the JavaTest harness is being used, writes the test harness configuration interview.
- **Quality Assurance Engineer**. Tests the TCK and documentation for accuracy, functionality, completeness, and performance.

- **Technical Writer**. Writes the TCK user guide and release notes. Works with a test developer to edit or revise the language of error messages and the test harness configuration interview. Helps to edit or revise the language of test specifications for readability.

Resource Estimates

The time it takes to complete your TCK depends on many factors specific to your technology and your TCK. Obviously, the scope of the TCK, the number of tests, the number of people you have working on it, the optional components, the tools you are using, and many other factors affect your schedule. There may also be specific platform limitations such as a small screen, memory limitations, or absence of a guaranteed way to perform some input on the device, that affect the time it takes to develop your TCK.

To assist your planning, time estimates based on TCK development experience at Sun are presented below. These estimates are given only for the purpose of illustrating a time-estimation method, **your own development metrics and time lines will certainly vary**.

As an initial general estimate, the experience at Sun indicates a rough equivalency between the time needed to develop the specification RI and the TCK. In other words, for every engineer-day spent on developing the RI, an engineer day is needed for the TCK.

4.5.1 Time Estimate Assumptions

The example TCK development estimates in the following sections are based on the following assumptions:

- The TCK is developed using the Java Compatibility Test Tools (Java CTT), including the JavaTest harness, as described in "Sun's Java Compatibility Test Tools" on page 9.
- Few functional tests are required. Most tests are API member tests.
- The specification has few methods or APIs that return different values depending on different environments, or on different implementations of the specification.
- The specification is not changing so rapidly that a significant number of tests have to be re-written in the middle of the development cycle.
- The API is not so complex that you have to go through a long chain of API calls to obtain an object to test.

Time Estimates—Test Development

The largest single component in TCK development is writing the TCK tests.

Obviously, the number of needed tests is governed by the quantity of packages, classes, methods, constructors, and fields in the API being tested and amount of specification assertion coverage desired.

The following test development time estimator may be useful in identifying the scope of effort required to build your test suite. This is based on experience within Sun for experienced TCK test development engineers. Your experiences will no doubt vary:

($A\text{-}Num \times CovBreadth \div DevRate$) $\times$ $NumEng$ = Test Development Time (in weeks)

Where:

- *A-Num* is the number of testable assertions defined by the specification.
- *CovBreadth* is the coverage breadth goal. For example, `0.75` for 75% breadth coverage.

  (Note that the ideal practice is to thoroughly test all elements which requires 100% breadth and depth coverage. Assertion breadth coverage of only 75% means that assertion testing is incomplete.)
- *DevRate* is the test development rate. According to Sun's experience a typical test development rate is approximately 12.5 test cases per week for a test developer to write, document, and debug assertion test cases (including simple test framework development).
- *NumEng* is the number of engineers writing tests.

Tip: If *A-Num* is not known, a rough approximation for *A-Num* can usually be estimated by multiplying the total number of public API members (in other words, all public methods, constructors, and fields) defined in the specification by 3. So if you have 1000 such public API members, you would estimate roughly 3000 testable assertions.

Using an *A-num* of 3000, a *CovBreadth* of 75%, and the estimated *DevRate* of 12.5 test cases per week per engineer, the above formula would return 180 weeks for test development time:

(3000 $\times$ .75% $\div$ 12.5) = 180 engineer-weeks

If you had 10 engineers writing and debugging tests, your test-development schedule would be 18 weeks.

Additional tests may also be needed for non-API features of the specification. This occurs when testing simple method, constructor, and field functionality is insufficient to fully verify the requirements of the specification. For example, functional testing may be needed to verify complex interactions between certain API members and classes. Estimating the number of such tests is difficult and highly dependent upon your specification.

Time Estimates—Other TCK Elements

Test writing is just one aspect of developing a full TCK. Time estimates for other aspects must be included as well:

- Reasonable time allotment for training the team to be able to properly read the specification and understand the testing environment needs to be added, typically: 1-2 months for a new engineer and 1-2 weeks for an experienced TCK test developer.
- Up to 25% of the estimated test development time should be included in your estimates for fixing bugs found by Quality Assurance evaluation of the TCK against the RI. This includes the time it takes to re-work tests invalidated by minor changes to the specification during the period of TCK development. QA testing can usually begin after the first stable build of the TCK.
- Two to four weeks for final QA approval of the TCK following completion of the test suite should also be included.
- While the test development estimation technique described above includes allowances for some simple test framework development, any major test framework requirements must be planned as well. For example, time and resource estimates should be included for:
  - Complex shared test library classes
  - Configurable test framework infrastructure (if needed) as well as any other code, examples, and/or tools necessary for the user to adapt the test harness to their own testing environments.
  - Writing and testing the means your customers will use to configure the TCK for use in their environment. For example, if you are using the JavaTest harness, the configuration interview questions and “more information” explanations have to be written, integrated, and tested.
  - Writing and testing a test agent and/or Java application manager if either are needed. (Note that Sun provides the JavaTest Agent that can be used as-is in many cases.)
- Time and resources must also be allocated to the development of the project build environment, such as writing makefiles, promoting and archiving builds, etc. This may vary significantly across different TCKs and development environments.
- Documentation time should be estimated for writing the user's guide, release notes, and working with the engineers on the test specifications (and a JavaTest harness configuration interview if one is used). Note that using the “Java Technology Compatibility Kit User's Guide Template” may significantly cut the time required to write your TCK user's guide.
- Reasonable time allotments must be added for several QA and development engineers to review the TCK documentation. It is suggested that they read and comment on at least two drafts of each document.
- Also plan for any additional time that may be required if the specification changes markedly during the period of TCK development, or if stable RI builds are unavailable for testing and debugging the tests.

APPENDIX **A**

TCK Appeals Process

This appendix describes the TCK appeals process, it contains the following sections:

Introduction

Under the JCP process, implementations of your technology specification must pass your TCK tests in order to be certified as compliant with that specification. Someone engaged in developing an implementation, however, may wish to challenge the fairness, validity, accuracy, or relevance of one or more of your TCK's tests. Therefore, the JCP process mandates a two-stage appeal process:

1. **First-level appeal**. The first-level appeals process allows those who are developing implementations of a technology specification to challenge one or more tests defined by the TCK. It is the responsibility of the Specification Lead to establish and determine how the first-level appeal process works. The details of the first-level appeals process must be described in your TCK documentation. Once the appeals process is established and the technology specification released, challenges are managed by the Maintenance Lead.

2. **Second-level appeal**. The second-level appeals process is mandated and managed by the appropriate Java Community Process Executive Committee (EC) to allow technology implementors who are not satisfied with a first-level decision to appeal it to the EC itself. The procedures for a second-level appeal are described in the *JCP: Process Document* (`http://jcp.org`).

   A second-level appeal should only be initiated in rare cases where the test challenger and Maintenance Lead have exhausted all other means of resolving the test issues.

A decision by a first or second-level appeal may require that the Maintenance Lead update or modify the TCK and/or the RI.

Appeals Process Guidelines

The first-level appeals process must be designed by the Specification Lead with assistance from the Expert Group to resolve TCK test challenges in a timely fashion.

At a minimum, a first-level appeals process should answer the following questions:

- Who can make challenges regarding the TCK?
- What kind of challenges to TCK test may be submitted? In other words, what are legitimate grounds for challenging a test? For example, some or all of the following might be considered as valid challenges to a test:
  - The test has bugs (for example, program logic errors).
  - The specification assertion covered by the test is ambiguous.
  - The test does not match the specification.
  - The test assumes unreasonable hardware and/or software requirements.
  - The test is biased for or against a particular implementation.
- What kind of challenges to TCK test are not accepted? In other words, what kinds of challenges are normally rejected? For example, challenges based upon issues unrelated to technical correctness, or challenges questioning the value of well defined specification features, may be rejected out of hand as inappropriate.
- How, and to whom, are these challenges submitted? What steps must someone take to challenge a TCK test?
- How, and by whom, are challenges decided? This might be accomplished by giving the Maintenance Lead complete authority to interpret the specification and rule on each test challenge, or there may be an evaluation committee, or some other mechanism for ruling on challenges.
- What happens if a challenge is accepted as valid? If using an Exclude List, how quickly is a challenged test added to the Exclude List and how is the new Exclude List made available to users?
- Can the Maintenance Lead create an alternative tests to address any challenge? If so, how is the alternate test made available?

  An alternate test might be permitted in situations where a problem is identified in one or more tests that are so critical to the testing process that their exclusion would jeopardize the integrity of the TCK. In these rare cases it may be necessary to make an alternate test available to TCK users that can be used in place of the invalid test. In other words, a TCK user could meet the conformance requirements by passing either the original test or its alternate. Note that alternate tests are quick fixes to problems that should be resolved in the next TCK release.

The following criteria are strongly encouraged in every first-level appeals process:

- All reviews for TCK challenges must follow the same process and be reviewed by either the Maintenance Lead or one or more designees of the Maintenance Lead.
- All implementors must be treated equally in terms of access to all TCK changes and updates including test corrections, additions and exclusions, pass/fail reporting criteria, and conformance requirements.

See the Java Technology Compatibility Kit User's Guide Template for an example of a first-level appeals process.

Test Appeal and Response Forms

Using designated forms for both a test challenge and a challenge rebuttal can facilitate the process of submitting test challenges and their related responses. The following examples include items that these forms might contain.

Example challenge form items:

- 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)

Example challenge rebuttal form items:

- Responder's name and company
- Responders role (for example, test developer or Maintenance Lead)
- Specification name(s) and version(s)
- Test suite name and version
- Exclude list version
- Test name
- Defense (argument for why test is valid)
- Alternatives (for example, are alternate tests appropriate?)
- Implications of test invalidity (for example, other affected tests and test framework code, creation or exposure of ambiguities in specification due to unspecified requirements, invalidation of the RI, creation of serious holes in the test suite, etc.)

APPENDIX **B**

TCK Project Planning Questions

This appendix describes the questions that various TCK planning documents need to address. It contains the following sections:

TCK Project Plan

The TCK Project Plan should normally answer questions such as:

- What is being tested by the TCK?
- What is the target level of test coverage that will be completed as of the product shipment date? In other words, if there are X number of assertions and API members, how many will be tested? At least 75% breadth assertion coverage is recommended. (See Appendix C, "Measuring TCK Test Coverage," for more information on test coverage.)
- What platforms will the Reference Implementation (RI) run on? Your TCK has to run against the RI on each RI platform.
- Which test harness (such as the JavaTest harness) will you use?
- What other tools (such as the Java CTT) will you use?
- What is the overall test architecture and design?
- Is any special hardware or software needed to run the tests?
- Will customers download the TCK from a web site, or will it be distributed by CD or other hard media?

- How will the TCK be assembled for customer delivery and installation? One common method is to distribute the TCK as a single ZIP file archive containing the test suite, test harness, and all associated documentation. Another possible method is to provide an installation program.
- Who is responsible for the various test development tasks?
- Who is responsible for ongoing maintenance of the tests?
- How long it will take to complete the TCK? (See "Resource Estimates" on page 43.)
- What are the milestones and schedule for test development and product development?
- What are the risks?

Preparation Plan

The Preparation Plan should normally answer questions such as:

- What resources are required to develop the TCK test suite? In addition to human resources, what equipment, software tools, space, support services, and other infrastructure will be needed?
- Will test suite engineers need training in conformance testing? If yes, what will that training consist of, and what resources will be required?
- What preliminary research and prototyping (if any) will be needed?

TCK Design Document

The TCK Design Document should normally answer questions such as:

- Which specification assertions need to be tested? (The list of assertions is the starting point of a test design plan.)
- Are there any items that will need to be included in the API implementation under test in order to execute the tests? For example, some special TCK related hardware or software might be required.
- Are there any issues that must be addressed due to a special TCK-related hardware or software characteristics? For example, support of a limited number of files or limited file name length.
- What specialized test framework code must be developed (if any)?
- Is a test agent needed? If yes, will you use the JavaTest Agent that comes with the JavaTest harness, or will you have to write your own agent? If a special test agent needs to be written, what are its requirements and parameters?

- Does the device being tested require Application Management Software as described in "Test Framework" on page 18? If so, is it available or does it have to be written? If it has to be written, what are its requirements and parameters?
- Are any enhancements required for the existing test development or test execution tools? For example, is there a need for unique or specialized hardware due to unusual characteristics or limitations of the technology under test?
- Are there any changes in the usual testing process that will be required by the planned test suite?
- What types of tests are being developed? Will they be differentiated based on class, API functionality, or other test classification characteristics? Ideally, the goal is to write both class-based and method/functionally-based tests for the entire implementation of the Java technology API specification.
- What test development will be proposed for the API? This needs to be described in detail, including sources for test development methodology and completeness measurements.
- What non-API tests need to be developed? Tests of classes, methods, constructors, and fields may not test everything in the specification that needs to be tested. For example, you may need to include tests of the security model, tests of how the application is deployed on a device, tests of the ways a server and a device communicate, tests of the ways in which the device reacts to the changes in a network environment, and so forth.
- How will the tests be incorporated into the test suite?
- How will the test suite be integrated with the test harness?
- How will test results be verified?
- How will areas of functionality to be tested be grouped into tests for the test suite?
- What performance criteria is required?
- Will any standard test-library code be needed?
- What resources will test development require (people, equipment, software, facilities, support)?

For more information see "Tests and the Test Suite" on page 23 and the *Java Technology Test Suite Development Guide.*

Test Specifications

Test specifications are human-readable descriptions in logical terms of what a test does and the expected results. (If adequate test specification information can be incorporated into the Test Design Document, then it is not necessary to develop exhaustive test specifications for each test in the kind of separate plan discussed here.)

Test specifications have a dual role. They are the plans that guide the writing of individual tests, and they provide the documentation describing each test which customers should receive as part of the TCK.

Therefore, in addition to describing what each test does, you may also need to address how the test specifications are delivered to your customers. For example, when using the JavaTest harness the recommended practice is to provide the individual test specifications in HTML within the test suite test directory tree. If they are straightforward and you are using HTML test description files with the JavaTest harness, the test specifications can be located within the same HTML file as the test description used to actually run the test; if not, test specifications can be placed be in separate HTML files with links to the appropriate test description files.

See "Tests and the Test Suite" on page 23 for additional information on test specifications.

Documentation Plan

The Documentation Plan should normally answer questions such as:

- What documentation will be supplied with the TCK? Typically, a TCK comes with a TCK user's guide, test harness user's guide, release notes, and test specifications.
- In what formats (HTML, PDF, ink-on-paper, etc,) will the documentation be delivered?
- What will be covered in the user's guide? (Typically a documentation plan includes a chapter outline.)
- Who will review the documentation? (Remember to include documentation review time in the reviewer's schedule and time estimates.)
- How long will it take for the documentation to be written, and reviewed?
- What is the documentation delivery schedule?
- Will your documents need to pass through legal or trademark reviews?
- Will they be copy-edited by someone other than the author?
- Will your documents comply with Section 508 of U.S. government accessibility requirements? If so, how will that be accomplished?
- Will your documents be localized? If so, to which other languages?
- What resources will documentation require (people, equipment, software, facilities, support)?

Note that as part of Java CTT, Sun provides a TCK user guide template that can be used as the basis for your user guide if you wish. For more information see "TCK Documentation" on page 28.

QA Plan

The QA Plan should normally answer questions such as:

- Are there any applicable standards used as references? For example, IEEE standard 829-1983.
- What test environment will be used?
- Which, or how many, builds will be tested?
- Which platforms will be tested?
- What is the pass/fail criteria?
- What will be tested and what testing approaches should be used? For example, how might the following elements be tested?
  - System as an entire entity
  - Individual components
  - TCK performance
  - TCK robustness
  - Generation of correct error messages
  - Test harness integration
  - TCK configuration options
  - TCK installation procedures
  - TCK user guide and other documentation
- How will the testing be designed and accomplished?
- What testing tools are needed? Do they already exist, or do they have to be created?
- How will problems and bugs be reported, tracked, and corrected?
- What are the QA testing time-line, schedule, and milestones?
- What resources will QA testing require (people, equipment, software, facilities, support)?

Testing a conformance test suite is a challenging task. While it is easy to verify that all TCK tests pass when run against a perfect RI, it is rare to have the luxury of “bad implementations” that can be used to trigger failures in your TCK tests. Therefore, it is important to identify QA testing approaches to verify as much “negative” testing as possible (in other words, creating situations that should cause TCK tests to report failures). Some ways to do this are to purposely misconfigure a TCK, construct RI implementations with known problems, or use specialized test framework code that can trigger failures in API tests. These testing approaches should be included in your QA plan.

APPENDIX C

Measuring TCK Test Coverage

This appendix describes metrics that you may wish to use in measuring TCK Test coverage. It contains the following sections:

Overview

A Technology Compatibility Kit (TCK) is a set of tests, tools and documentation that allows an implementor of an associated Java Technology Specification to determine if the implementation is compliant with the Specification. To accomplish its purpose, the TCK test suite must adequately cover the entire specification.

To assess the quality of your TCK, and to ensure that the entire Specification has been adequately covered, Sun recommends addressing the following metrics:

- "Assertion Coverage" on page 55
- "API Coverage" on page 59
- "Signature Coverage" on page 60

Terminology

The following terms are used in this Appendix:

- **API member**. Fields, methods and constructors for all public classes that are defined in the Specification.
- **Assertion**. A specific statement of functionality or behavior derived from a specification.

  For example: from the method description for `java.lang.Integer.toString(int i, int radix)`:

```
If the radix is smaller than Character.MIN_RADIX or larger than
 Character.MAX_RADIX, then the radix 10 is used instead
```

From the method description for `java.lang.Integer.parseInt(Strings)`:

```
Throws: NumberFormatException if the String does not contain a
 parsable int
```

- **Assertion attribute**. One or more binary true/false values that are applied to an assertion. (See "Assertion Attributes" on page 54 and TABLE 2 on page 55 for more information.)
- **Test case**. A single set of test inputs, execution conditions, and expected results developed to verify an implementation's conformance with a specific assertion.

  Note that multiple test cases will often be required to thoroughly test assertion conformance. (Equivalence class partitioning and boundary value analysis techniques are typically used to identify suitable test cases.)

  For example, the `java.lang.Integer.toString` assertion quoted above would be tested by passing several different values for the radix parameter and verifying that the specified behavior resulted. Logical groupings of values (for example two or three between `Character.MIN_RADIX` and `Character.MAX_RADIX`) might be applied in a single test case.

Assertion Attributes

Each of the attributes listed in TABLE 2 on page 55 are applied to every assertion. These attributes have binary ('true' or 'false') values. Note that more than one attribute may be true for a particular assertion.

TABLE 2 Assertion Attributes

Attribute	TRUE	FALSE
Required NOTE: Typically, optional features are nevertheless tightly specified. That is, implementors may be free not to implement a feature, but if they do so they must follow the specification.	The functionality specified in the assertion must be implemented. (This is typically indicated in the specification by use of the words “must” or “shall”.)	Implementation is optional (typically indicated by use of the word “may”).
Implementation-specific	Precise details of the behavior are deliberately unspecified; how they are implemented is left to the discretion of the implementor. By definition, such behavior is untestable for the purpose of conformance testing.	The behavior is specified, and therefore (in principle) is testable.
Testable	This assertion can be tested.	This assertion can not be tested, because the preconditions or the test observation cannot be achieved reliably or portably, because creating or executing tests would require an unreasonable amount of time or resources, or because the assertion is ambiguous (see next attribute).
Ambiguous	The assertion is ambiguous, and therefore cannot be tested. (Typically this will be considered a bug in the specification.)	The assertion is unambiguous, and therefore (in principle) is testable.

Assertion Coverage

This section discusses assertion coverage and the processes and metrics used to measure it.

Process

1. **Specification mark-up**. You review the specification, identifying assertions and classifying them by assigning the appropriate values to each assertion attribute. For example, “Is the assertion testable, or ambiguous, yes or no?” Sun's Spec Trac tool significantly aides in this mark-up process.

2. **Assertions to be tested**. You decide which assertions should be tested, and to what level of detail. The marked-up specification should be reviewed and approved by the Specification Lead.

   It is likely and appropriate that different portions of the specification will be tested with different levels of thoroughness. You should select for detailed testing those portions of the specification that are most critical for conformance.

   Specific issues are presented by "external specifications" referenced by Java specifications. See "External Specifications" below for a more detailed discussion of this problem.

3. **Tests**. You develop test cases for the assertions to be tested. Test cases must be identified and associated with the assertions they test.

4. **Coverage is evaluated.** Assertion coverage metrics are calculated. If Sun's Spec Trac Tool is used for specification mark-up and test identification, and you first bind assertions to test cases, these metrics can be calculated automatically.

External Specifications

You may wish to make a distinction between *Java specifications* and *external specifications*. External specifications are typically from independent standards bodies that are referenced by Java specifications. Often a relatively "thin" Java API is provided, but reference is made to "dense" external specifications. For example, the relatively small JAXP API references and in some sense includes the extremely complex W3C XML language specifications.

Whether or not these external specifications should be marked up for assertion classification with the same rigor as core Java specifications, and the thoroughness with which the assertions derived from them should be tested, will depend on the precise language used in the specification.

Specification leads should choose their language carefully, lest it inadvertently impose significant TCK test development requirements.

Consider as an example a Java API to invoke a compiler for a complex language that is defined in external specification. The Java API could be written in an "exclusive" or an "inclusive" manner.

An inclusive specification might state:

```
When invoked via the 'compile(file)' method, the compiler reads
the file, which is assumed to contain a program in 'X' language.
If the program is syntactically correct the method returns true
else it returns false.
```

This statement should probably be interpreted as two assertions (one for syntactically correct, and one for incorrect program files). However, the appropriate number of test cases for each assertion is likely to be many thousands for even a

moderately complex language, since a full range of valid and invalid inputs should be tested. The identification and construction of these test cases would require a complete mark-up of the external specification.

Conversely, an exclusive specification might state:

```
When invoked via the compile(file) method, the compiler reads the
file, which is assumed to contain a program in 'X' language. If
the compiler determines that the program is syntactically correct
the method returns true else it returns false. How the compiler
makes that determination, and whether or not it does so in
conformance with the specifications for the 'X' language is
outside the bounds of this specification.
```

In this situation, a relatively small number of test cases should be adequate to verify that the invoking method has been constructed properly, however, it would not be necessary to exhaustively test the full range of valid and invalid program file inputs.

It might be advisable under these circumstances to reference an external conformance test suite that could be used to verify the correct behavior of the compiler.

Assertion Coverage Metrics

The following metrics can be calculated for each package in the technology implementation.

- "Specification Testability" on page 57
- "Assertion Coverage (Breadth)" on page 58
- "Assertion Coverage (Depth)" on page 58

Note – Any tests that are delivered with a TCK, but are excluded from test runs, must also be excluded from coverage calculations.

Specification Testability

Specification testability measures how testable the specification is, and the extent to which it encourages application portability.

If the specification is so ambiguous or incomplete that a significant proportion of its assertions cannot be tested, even high assertion coverage of those assertions that are testable cannot promote conformance among implementations.

Similarly, while it is possible to test implementations of specifications that contain significant portions of optional functionality, real-world applications are unlikely to be portable across such implementations due to the different sets of implemented features.

Metrics

- **Total assertion**s = total number of identified assertions derived from the specification.
- **Implementation-specific assertions** = number of identified assertions that are classified as implementation-specific.
- **Ambiguous assertions** = number of identified assertions that are classified as ambiguous.
- **Testable assertions** = number of identified assertions that are classified as testable.

Testability ratio = testable assertions/total assertions expressed as a percentage

Based on this formula, your TCK Project Plan might set a target testability ratio of at least 70% averaged across all packages.

Assertion Coverage (Breadth)

Assertion coverage (breadth) measures how broadly the TCK tests the implementation by calculating how many of the identified, testable assertions are actually tested at least once, no matter how superficially.

Metric

- **Testable assertions** = the number of identified, testable assertions derived from a portion of the specification.
- **Tested assertions** = the number of those assertions for which at least one test case is provided.

Breadth ratio = tested assertions / testable assertions expressed as a percentage

Based on this formula, your TCK Project Plan might set a target of the following minimums:

- Averaged across all packages a breadth ratio of at least 75%.
- For each package a minimum breadth ratio of at least 30%.

Assertion Coverage (Depth)

Assertion coverage (depth) measures the completeness with which the TCK tests the implementation, by calculating the number of test cases actually provided as a proportion of the total number of test cases that would be required to thoroughly test the implementation.

For information on how to estimate the number of test cases required to thoroughly test an implementation, see the discussions of Equivalence Class Partitioning and Boundary Value Analysis in *Java Technology Test Suite Development Guide*.

Metric

- **Testable assertions** = the number of identified, testable assertions derived from a portion of the specification.
- **Provided test cases** = the total number of test cases provided to test those assertions.
- **Required test cases** = the total number of test cases that would be required to thoroughly test those assertions.
- **Test cases per assertion** = provided test cases / testable assertions.

Depth ratio = provided test cases / required test cases, expressed as a percentage

API Coverage

This section discusses API member coverage and the processes and metrics used to measure it.

Process

API coverage, while not as useful as assertion coverage for determining TCK completeness, provides information about the percentage of API members that are directly referenced by the TCK. This measurement is determined by counting the number of public API members (that is, public methods, constructors and fields) directly referenced by the TCK tests and comparing them to the total number of public API members defined by the specification.

API Coverage Metric

The API coverage metric measures how many API members are directly referenced at least once by the TCK tests. Ideally, all API members should be accessed at least once by the TCK tests meaning that all API members defined by the Specification are tested at least marginally by the TCK.

Metric

- **Accessed API members** = number of API members referenced by TCK tests.
- **Spec API members** = number of API members defined by the Specification.

API coverage ratio = accessed API members/ Spec API members, expressed as a percentage

Based on this formula, your TCK Project Plan might set a target for API coverage ratio. A ratio of at least 90% is recommended.

Signature Coverage

The signature coverage metric ensures that all required elements in the specification are present in the API and that no extra elements are present that improperly extend the API.

The recommended target is 100% of signatures verified.

Java TCK and CTT Glossary

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

active agent — A type of *test agent* that initiates a connection to the *JavaTest harness*. Active test agents allow you to run tests in parallel using many agents at once and to specify the test machines at the time you run the tests. Use the *agent monitor* to view the list of registered active agents and synchronize active agents with the JavaTest harness before running tests. See also *test agent*, *passive agent*, and *JavaTest agent*.

active applet instance — An applet instance that is selected on at least one of the logical channels.

agent monitor — The JavaTest window that is used to synchronize *active agents* and to monitor agent activity. The Agent Monitor window displays the agent pool and the agents currently in use.

agents — See *test agent*, *active agent*, *passive agent*, and *JavaTest agent*.

all values — All of the *configuration values* required for a test suite. All values includes the test environment values specific to that test suite and the JavaTest *standard values*.

API member — Fields, methods and constructors for all public classes that are defined in the specification.

API member tests — Tests (sometimes referred to as *class and method* tests) that are designed to verify the semantics of API members.

appeals process — A process for challenging the fairness, validity, accuracy, or relevance of one or more TCK tests. Tests that are successfully challenged are either corrected or added to the TCK's *Exclude List*. See also *first-level appeals process*, *second-level appeals process*, and *Exclude List*.

Application IDentifier (AID) An identifier that is unique in the TCK *namespace*. As defined by ISO 7816-5, it is a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies. There is a unique AID for each package and a unique AID for each applet in the package. The package AID and the default AID for each applet defined in the package are specified in the CAP file. They are supplied to the converter when the CAP file is generated.

Application Programming Interface (API) An API defines calling conventions by which an application program accesses the operating system and other services.

Application Protocol Data Unit (APDU) A script that gets sent to the test applet as defined by ISO 7816-4.

assertion A statement contained in a structured Java technology API specification to specify some necessary aspect of the API. Assertions are statements of required behavior, either positive or negative, that are made within the *Java technology specification*.

assertion testing Conformance testing based on testing assertions in a specification.

atomic operation An operation that either completes in its entirety (if the operation succeeds) or is not completed at all (if the operation fails).

automatic tests Test that run without any intervention by a user. Automatic tests can be queued up and run by the *test harness* and their results recorded without anyone being present.

behavior-based testing A set of test development methodologies that are based on the description, behavior, or requirements of the system under test, not the structure of that system. This is commonly known as “black-box” testing.

boundary value analysis A test development technique which entails developing additional tests based on the boundaries defined by previously categorized equivalence classes.

class The prototype for an object in an *object-oriented* language. A class may also be considered a set of objects which share a common structure and behavior. The structure of a class is determined by the class variables which represent the state of an object of that class and the behavior is given by a set of methods associated with the class. See also *classes*.

classes Classes are related in a class hierarchy. One class may be a specialization (a “subclass”) of another (one of its “superclasses”), may be composed of other classes, or may use other classes in a client-server relationship. See also *class*.

configuration Information about your computing environment required to execute a *Technology Compatibility Kit (TCK)* test suite. The *JavaTest harness* version 3.x uses a *configuration interview* to collect and store configuration information. The *JavaTest harness* version 2.x uses *environment files* and *parameter files* to obtain configuration data.

configuration editor The dialog box used by *JavaTest harness* version 3.x to present the *configuration interview*.

configuration interview A series of questions displayed by the *JavaTest harness* version 3.x to gather information from the user about the computing environment in which the TCK is being run. This information is used to produce a *test environment* that the JavaTest harness uses to execute tests.

configuration value Information about your computing environment required to execute a TCK test or tests. The *JavaTest harness* version 3.x uses a *configuration interview* to collect configuration values. The JavaTest harness version 2.x uses *environment files* and *parameter files* to obtain configuration data.

conformance rules Define the criteria a Java technology implementation must meet in order to be certified as "conformant" with the technology specification. See also *conformance testing*.

conformance testing The process of testing an implementation to make sure it is conformant with the corresponding Java technology specification. A suite of tests contained in a *Technology Compatibility Kit (TCK)* is typically used to test that the implementation meets and passes all of the *conformance rules* of that specification.

environment files Files used by the *JavaTest harness* 2.x to configure how the JavaTest harness runs the tests on your system. Environment files have the filename extension of `.jte`. Environment files are used in conjunction with *parameter files*.

equivalence class partitioning A test development technique which entails breaking a large number of tests into smaller subsets with each subset representing an equivalent category of *test*.

Exclude List A list of TCK tests that a *technology implementation* is not required to pass in order to certify conformance. The *JavaTest harness* uses exclude list files (`*.jtx`), to filter out of a test run those tests that do not have to be passed. The exclude list provides a level playing field for all implementors by ensuring that when a test is determined to be invalid, no implementation is required to pass it. Exclude lists are maintained by the *Maintenance Lead (ML)* and are made available to all technology licensees. The ML may add tests to the exclude list for the test suite as needed at any time. An updated exclude list replaces any previous exclude lists for that test suite.

first-level appeals process The process by which a technology implementor can appeal or challenge a TCK test. First-level appeals are resolved by the Expert Group responsible for the technology specification and TCK. See also *appeals process* and *second-level appeals process*.

framework See *test framework*.

Graphical User Interface (GUI) Provides application control through the use of graphic images.

HTML test description A *test description* that is embodied in an HTML table in a file separate from the test source file.

implementation See *technology implementation*.

instantiation In object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

interactive tests Tests that require some intervention by the user. For example, the user might have to provide some data, perform some operation, or judge whether or not the implementation passed or failed the test.

Java™ 2, Standard Edition (J2SE™) platform A set of specifications that defines the desktop runtime environment required for the deployment of Java applications. The J2SE implementations are available for a variety of platforms, but most notably the Sun Solaris operating environment and Microsoft Windows.

Java Application Manager (JAM) A native application used to download, store and execute Java applications.

Java Archive (JAR) A platform-independent file format that combines many files into one.

Java Compatibility Test Tools (Java CTT) Tools, documents, templates, and samples that can be used to design and build TCKs. Using the Java CTT simplifies conformance test development and makes developing and running tests more efficient.

Java Community Process (JCP) An open organization of international Java developers and licensees whose charter is to develop and revise *Java technology specifications*, and their associated *Reference Implementation (RI)*, and *Technology Compatibility Kit (TCK)*.

Java Platform Libraries The class libraries that are defined for each particular version of a *Java technology* in its *Java technology specification*.

Java Specification Request (JSR) The actual descriptions of proposed and final technology specifications for the Java technology.

Java technology A Java technology is defined as a *Java technology specification* and its *Reference Implementation (RI)*. Examples of Java technologies are Java 2 Platform, Standard Edition (J2SE™), the Connected Limited Device Configuration (CLDC), and the Mobile Information Device Profile (MIDP).

Java Technology Compatibility Kit See *Technology Compatibility Kit (TCK)*.

Java technology specification A written specification for some aspect of the *Java technology*.

JavaTest agent A *test agent* supplied with the *JavaTest harness* to run TCK tests on a Java implementation where it is not possible or desirable to run the main JavaTest harness. See also *test agent*, *active agent*, and *passive agent*.

JavaTest harness A *test harness* that has been developed by Sun to manage test execution and result reporting for a *Technology Compatibility Kit (TCK)*. The harness configures, sequences, and runs test suites. The JavaTest harness is designed to provide flexible and customizable test execution. It includes everything a test architect needs to design and implement tests for Java technology specifications.

keywords Used to direct the *JavaTest harness* to include or exclude tests from a test run. Keywords are defined for a *test* by the *test suite* architect.

Maintenance Lead (ML) The person responsible for maintaining an existing *Java technology specification* and related *Reference Implementation (RI)* and *Technology Compatibility Kit (TCK)*. The ML manages the TCK *appeals process*, *Exclude List*, and any revisions needed to the specification, TCK, or RI.

methods Procedures or routines associated with one or more *classes*, in *object-oriented* languages.

MultiTest A JavaTest library class that enables tests to include multiple test cases. Each test case can be addressed individually in a test suite *Exclude List*.

namespace A set of names in which all names are unique.

object-oriented A category of programming languages and techniques based on the concept of *objects* which are data structures encapsulated with a set of routines, called *methods*, which operate on the data.

objects In *object-oriented* programming, objects are unique instances of a data structure defined according to the template provided by its *class*. Each object has its own values for the variables belonging to its class and can respond to the messages (*methods*) defined by its class.

packages A *namespace* within the Java programming language. It can have *classes* and interfaces. A package is the smallest unit within the Java programming language.

parameter files Files used by the *JavaTest harness* to configure individual test runs. For JavaTest harness version 2.x parameter files have the filename extension `*.jtp`. For JavaTest harness version 3.x parameter files have the filename extension `*.jti`.

passive agent A type of *test agent* that must wait for a request from the *JavaTest harness* before they can run tests. The JavaTest harness initiates connections to passive agents as needed. See also *test agent*, *active agent*, and *JavaTest agent*.

prior status A JavaTest filter used to restrict the set of tests in a test run based on the last test result information stored in the test result files (`.jtr`).

Profile Specification A specification that references one of the Platform Edition Specifications and zero or more other Java technology specifications (that are not already a part of a Platform Edition Specification). APIs from the referenced Platform Edition must be included according to the referencing rules set out in that Platform Edition Specification. Other referenced specifications must be referenced in their entirety.

Program Management Office (PMO) The administrative structure that implements the *Java Community Process (JCP)*.

Reference Implementation (RI) The prototype or proof of concept implementation of a *Java technology specification*. All new or revised specifications must include an RI. A specification RI must pass all of the TCK tests for that specification.

second-level appeals process Allows technology implementors who are not satisfied with a first-level appeal decision to appeal the decision. See also *appeals process* and *first-level appeals process.*

signature file A text representation of the set of public features provided by an API that is part of a finished TCK. It is used as a signature reference during the TCK signature test for comparison to the technology implementation under test.

signature test Checks that all the necessary API members are present and that there are no extra members which illegally extend the API. It compares the API being tested with a reference API and confirms if the API being tested and the reference API are mutually binary compatible.

specification See *Java technology specification*.

standard values A *configuration value* used by the JavaTest harness to determine which tests in the test suite to run and how to run them. The user can change standard values using either the *all values* or Standard Values view in the *configuration editor*.

structure-based testing A set of test development methodologies that are based on the internal structure or logic of the system under test, not the description, behavior, or requirements of that system. This is commonly known as “white-box” or “glass-box” testing. Conformance testing does not make use of structure-based test techniques.

system configuration Refers to the combination of operating system platform, Java programming language, and JavaTest harness tools and settings.

tag test description A *test description* that is embedded in the Java language source file of each test.

TCK coverage file Used by the Java CTT Spec Trac tool to track the test coverage of a *test suite* during test development.

Technology Compatibility Kit (TCK) The suite of tests, tools, and documentation that allows an implementor of a *Java technology specification* to determine if the implementation is compliant with the specification.

technology implementation Any binary representation of the form and function defined by a *Java technology specification*.

technology specification See *Java technology specification*.

test agent A Java application that receives tests from the *test harness*, runs them on the implementation being tested, and reports the results back to the test harness. Test agents are normally only used when the TCK and implementation being tested are running on different platforms. See also *active agent*, *passive agent*, and *JavaTest agent*.

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

test cases A small test that is run as part of a set of similar *tests*. Test cases are implemented using the JavaTest *MultiTest* library class. A test case tests a specification assertion, or a particular feature, or part of a feature, of an assertion.

test command A class that knows how to execute test classes in different environments. Test commands are used by the *test script* to execute tests.

test command template A generalized specification of a *test command* in a *test environment*. The test command is specified in the test environment using variables so that it can execute any test in the test suite regardless of its arguments.

test description Machine readable information that describes a test to the *test harness* so that it can correctly process and run the related test. The actual form and type of test description depends on the attributes of the *test suite*. A test description exists for every test in the test suite and is read by the *test finder*. When using the *JavaTest harness*, the test description is a set of test-suite-specific name/value pairs in either HTML tables or Javadoc-style tags.

test environment Consists of one or more *test command template* that the *test script* uses to execute tests and set of name/value pairs that define *test description* entries or other values required to run the tests.

test execution model The steps involved in executing the tests in a *test suite*. The test execution model is implemented by the *test script*.

test finder When using the *JavaTest harness*, a nominated class, or set of classes, that read, verify, and process the files that contain *test description* in a *test suite*. All test descriptions that are located or found are handed off to the JavaTest harness for further processing.

test framework Software designed and implemented to customize a test harness for a particular test environment. In addition to the *test harness*, a test framework might (or might not) include items such as a: *configuration interview*, *Java Application Manager (JAM)*, *test agent*, *test finder*, *test script*, and so forth. A test framework might also include user-supplied software components (plug-ins) to provide support for implementation-specific protocols.

test harness The applications and tools that are used for test execution and *test suite* management. The *JavaTest harness* is an example of a test harness.

test script A Java class whose job it is to interpret the *test description* values, run the tests, and then report the results back to the *JavaTest harness*. The test script must understand how to interpret the test description information returned to it by the *test finder*.

test specification A human-readable description, in logical terms, of what a test does and the expected results. Test descriptions are written for test users who need to know in specific detail what a test does. The common practice is to write the test specification in HTML format and store it in the *test suite's* test directory tree.

test suite A collection of tests, used in conjunction with the *test harness* to verify conformance of the licensee's implementation to a *Java technology specification*. Every *Technology Compatibility Kit (TCK)* contains one or more test suites.

work directory A directory associated with a specific test suite and used by the *JavaTest harness* to store files containing information about the test suite and its tests.

Index

Q

R

S

T

W