SigTest User's Guide

Version 2.1

Sun Microsystems, Inc.
www.sun.com

February 2009

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

Please
Recycle

Contents

Tables

Code Examples

Preface

This guide describes how to install and run the SigTest collection of tools. This collection includes the Signature Test tool and the API Coverage tool. Signature Test tool includes utilities used to develop signature test components that can be used to compare API test signatures. API Coverage tool is used to estimate the test coverage a test suite provides for an implementation of a specified API.

Note – For simplicity, this user's guide refers to the test harness as the *JavaTest harness*. Note that the open source version of the harness, called *JT harness*, can be used in its place. The JT harness software can be downloaded from: http://jtharness.dev.java.net/

Who Should Use This Guide

This guide is for developers of quality assurance test suites and developers of compatibility test suites — TCKs for a Java™ platform API as part of the Java Community Process™ (JCP™) program.

Before You Read This Guide

Before reading this guide, it is best to be familiar with the Java programming language. A good resource for the Java programming language is the Sun Microsystems, Inc. web site, located at http://java.sun.com.

Note – Web URLs provided are subject to change.

How This Guide Is Organized

Introduction describes the SigTest collection of tools.

Part I describes how you can use the *Signature Test tool* to easily compare the signatures of two different implementations of the same API.

Part II describes how you can use the *API Coverage tool* to estimate the test coverage a test suite provides for an implementation of a specified API.

Part III contains an appendix that includes step-by-step examples that show how to use Signature Test tool.

Related Documentation

For details about the Java programming language, see the following documents:

- *The Java Programming Language, Third Edition*
- *The Java Language Specification, Second Edition*
- *The Java Virtual Machine Specification, Second Edition*

These documents are available at `http://java.sun.com/docs/books/`.

Typographic Conventions

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

`\` or `^`	A backslash at the end of a line indicates that a long code line has been broken in two on a UNIX® system, typically to improve legibility in code. The caret character (^) indicates this on a Microsoft Windows system.	`java classname \`
		`[`*classname_arguments*`]`
		
		`java classname ^`
		`[`*classname_arguments*`]`

Indented code or command line	Indicates a wrapped continuation from a previous line with no carriage return or return character in the actual code.	java classname `[`*classname_arguments*`]`

* The top-most SigTest Tool collection installation directory is referred to as *SigTest-Directory* throughout the SigTest Tool collection documentation.

Sun Welcomes Your Comments

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

CHAPTER 1

Introduction

The SigTest product is a collection of tools that can be used to compare APIs and to measure the test coverage of an API. The tools were originally created to assist in the creation of Java technology compatibility test suites (TCKs), but are also useful in the creation of other types of test suites and in the software development process. The SigTest product consists of the following two tools.

The **Signature Test tool** makes it easy to compare the signatures of two different implementations or different versions of the same API. When it compares different implementations of the same API, the tool verifies that all of the members are present, reports when new members are added, and checks the specified behavior of each API member. When it compares different versions of the same API, the tool checks that the old version can be replaced by the new one without adversely affecting existing clients of the API.

The **API Coverage tool** can be used to estimate the test coverage a test suite provides for an implementation of a specified API. It does this by determining how many public class members the test suite references within the API specification. The tool uses a signature file representation of the API specification as the source of specification analysis. It does not process a formal specification in any form.

Part I of this manual describes how to use the Signature Test Tool, Part II describes how to use the API Coverage tool and Part III contains two appendices that include step-by-step examples that show how to use the Signature Test tool and the rules used in the Signature Test API migration feature.

PART I Signature Test Tool

This part describes how you can use the *Signature Test Tool* to easily compare the signatures of two different implementations or different versions of the same API.

CHAPTER 2

Introduction to Signature Test Tool

You can use the Signature Test tool to easily compare the signatures of two different implementations of the same API. It verifies that all of the members are present, reports when new members are added, and checks the specified behavior of each API member.

Signature Test Basics

A signature test compares two implementations of an API and reports the differences. The Signature Test tool compares the signatures of two implementations or versions of the same API and can do the following:

- Create and run a test that verifies that all of the members are present
- Report when new members are added
- Check the specified behavior of each API member
- Determine whether the old version of the API can be replaced with the newer one without adversely affecting existing clients of the API

The signature test created by the Signature Test tool can be run independently at the command line, or under the control of the JavaTest™ harness.

Note – For simplicity, this user's guide refers to the test harness as the *JavaTest harness*. Note that the open source version of the harness, called *JT harness*, can be used in its place. The JT harness software can be downloaded from: `http://jtharness.dev.java.net/`

The Signature Test tool was originally created to assist in the creation of Java technology compatibility test suites (TCKs). It simplified the process of verifying that the API signature of a new implementation of a Java technology matched the signature of a reference implementation of that API.

When used in a software development environment, the Signature Test tool can be used to track and control changes to an API throughout the development process.

What is Tested

The signature test algorithm compares the API implementation under test with a signature file created from the API you are comparing it to — often referred to as a *reference implementation*. The signature test checks for compatibility by verifying the equality of API member sets. By checking for mutual compatibility of API member sets, the test verifies that the following conditions are true:

- If an API item is defined in the reference implementation of the API, then that item is implemented in the API under test, and vice versa.
- Attributes chosen for comparison are identical in both implementations of the API. The tool chooses attributes for comparison according to the type of check being processed. This is described more in "Mutual Binary Compatibility Check" on page 7 and "Mutual Source Compatibility Check" on page 7.

By checking for migration compatibility of API member sets, the test verifies that the following conditions are true:

- If an API item is defined in the previous version of the API, then that item is implemented in the API version under test
- If an API item is not defined in the previous version of the API, but added to the API version under test, the added item does not break backward compatibility. This is described more in "Migration Binary and Source Code Check" on page 7.
- Attributes chosen for comparison are identical in both versions of the API or their change does not break backward compatibility. The tool chooses attributes for comparison according to the type of check being processed. This is described more in "Migration Binary and Source Code Check" on page 7.

Mutual Binary Compatibility Check

The signature test binary compatibility check mode verifies that a Java technology implementation undergoing compatibility testing and its referenced APIs are mutually binary compatible as defined in Chapter 13, "Binary Compatibility," of *The Java Language Specification*. This assures that any application runs with any compatible API without any linkage errors.

This check is less strict than the default source compatibility check, described next. It is for use primarily in the special case of when a technology is developed for Java technology environments that are purely runtime. Such an environment does not provide a Java technology-based compiler (Java compiler), nor does it include class files that could be used to compile applications for that environment. Because of the limited use of such an environment, the API requirements are slightly relaxed compared to environments that support application development.

Java application environments can contain several Java technologies. Not all Java technologies can be combined with each other, and in particular, their sets of API signatures might be incompatible with each other. Relaxing signature checks to the level of mutual binary compatibility allows the developer to combine technologies in a purely runtime environment that cannot be combined otherwise.

Mutual Source Compatibility Check

While binary compatibility is important, it cannot guarantee that an application in binary form as a set of class files can be *recompiled* without error.

The signature test source compatibility check mode verifies that any application that compiles without error with a compatible API, compiles without error with all other source compatible APIs.

Mutual source compatibility is a stricter check than the mutual binary compatibility and the Signature Test tool performs it by default.

Migration Binary and Source Code Check

Mutual compatibility is generally used in certification processes where the goal is to ensure that an alternative or third-party implementation of an API conforms to a reference implementation. Application developers have a different concern, they must ensure that evolving library APIs that their applications link to continue to work with customers' applications. The Signature Test tool can be used to check APIs as they evolve and ensure both binary and source code migration compatibility.

Migration binary compatibility checking ensures that there will be no linkage errors between pre-existing client binaries and the new version. This determination is based on the Chapter 13, "Binary Compatibility," of The Java Language Specification.

Migration source code compatibility means that pre-existing client source code can be recompiled with the new version without compilation errors.

Class and Class Member Attributes Checked

A Java platform API consists of classes, and interfaces, and their member fields, methods, and constructors, and documented annotations. In turn, all of these API items can have various attributes such as names, modifiers, a list of parameters, a list of interfaces, exceptions, nested classes, and so forth. A signature test checks that certain members and attributes belonging to the API under test are the same as those defined by the API to which it is being compared. The signature test checks public and protected API items only and ignores private and package-access items.

The tool checks the following attributes when comparing items in the API implementation under test:

- Classes and interfaces, including nested classes and interfaces:
  - Set of modifiers except `strictfp`
  - Name of the superclass
  - Names of all superinterfaces, direct plus indirect, where order is insignificant
- Constructors:
  - Set of modifiers
  - List of argument types
  - In source compatibility mode only, the normalized list of thrown exceptions where order is insignificant

    Normalizing the throw lists involves removing all superfluous exception classes. An exception class is superfluous if it is a subclass of either the `java.lang.RuntimeException` class, the `java.lang.Error` class, or another class from the same list.
- Methods:
  - The set of modifiers, except `strictfp`, `synchronized`, and `native`
  - The return type
  - The list of argument types
  - In source mode only, the normalized list of thrown exceptions, described earlier, where order is insignificant
- Fields:

- Set of modifiers, except `transient`
- Field type

- Documented annotations with `SOURCE` and `RUNTIME` retention of the following types:
  - Classes and interfaces
  - Fields, methods and constructors
  - Parameters and annotation types

The tool performs the check in the following order:

1. For all top-level public and protected classes and interfaces, it compares the attributes of any classes and interfaces with the same fully qualified name.
2. Taking into account all declared and inherited members, it compares all public and protected members of the same kind and same simple name, treating constructors as class members for convenience sake.

Source and Binary Compatibility Modes

Earlier Signature Test tool versions performed a comparison of all exceptions declared in `throws` clauses for methods and constructors. Certain variations in this area caused an error message during the signature test. Despite these error messages, the source files compiled successfully together. Successful compilation is the basic criteria for source compatibility with the current Signature Test tool, while successful linking as the basic criteria for binary compatibility.

Changes to the `throws` clause of methods or constructors do not break compatibility with existing binaries because these clauses are checked only at compile time, causing no linkage error. For the purpose of signature testing, this relates directly to binary compatibility as described earlier in "Mutual Binary Compatibility Check" on page 7.

The adaptation of JSR 68, *The Java ME Platform Specification*, formalized the use of building blocks in API development. A building block is a subset of an existing API that is approved for reuse in the construction of profiles or optional packages. The building block concept enables a developer to duplicate the functionality provided by another API without having to redefine an entirely new API. For further details see JSR 68 at
`http://www.jcp.org/en/jsr/detail?id=68`.

The use of building blocks created a need for more lenient checking of exception throw lists compared to earlier Signature Test tool versions. Consequently, Signature Test tool 2.1 provides both a source and a binary compatibility mode of operation. This retains compatibility with earlier signature files while adding support for building blocks and eliminating the unnecessary error messages.

The `SignatureTest` command recognizes the `-mode` option that takes the values "src" or "`bin`" as arguments for choosing source mode or binary mode. The choice of which mode to use depends on the type of signature file being used in the test. This is described in more detail later in these sections:

- "`Setup` Command" on page 20 describes how to generate a signature file
- "`SignatureTest` Command" on page 27 describes how to specify the mode when running a signature test
- "`Merge` Command" on page 37 describes how to generate a combined signature file from set of signature files

The difference between the binary and source compatibility modes is how the tool handles the `throws` list for constructors and methods (as described in "Class and Class Member Attributes Checked" on page 8). Constant checking behavior is also different in binary and source compatibility modes. Although constant checking can be applied to binary compatibility, it is a necessary prerequisite for source code compatibility. "Constant Checking in Differing Run Modes" on page 15 describes these differences in more detail.

Using Custom Signature Loaders

The signature test has a requirement for the Java Platform, Standard Edition (Java SE platform) runtime environment version 1.4 or later. This requirement might prevent use of the tool on limited or nonstandard environments such as some Java Platform, Micro Edition (Java ME platform) or Java Platform, Enterprise Edition (Java EE platform) configurations.

To overcome this, the tool provides support for custom signature loaders that can be implemented as plug-ins. These plug-ins gather signatures from a runtime environment when the `SignatureTest` command cannot be run directly. For example, you might create a light-weight remote JavaTest harness agent and run the signature loader on a remote Connected Device Configuration (CDC) compatible device. Another example is using a wrapped J2EE platform bean as a signature loader inside a J2EE platform container where any direct file I/O operations are prohibited.

As an aid in developing such an extension, the Signature Test tool distribution includes a class library that contains a signature serializer and some related utility classes in the *SigTest-Directory*`/lib/remote.jar` file. This file contains a subset of the Signature Test tool classes that are necessary to develop a custom plug-in. All of these library classes are CDC 1.0 compatible and have minimal memory requirements. The source code for these classes is distributed in the *SigTest-Directory*`/redistributables/sigtest_src.zip` file. The code is designed for running a plug-in with the JavaTest harness using the Java ME Framework. The server and client source code and the HTML test descriptions for an actual plug-in example are located in the *SigTest-Directory*`/examples/remote` directory.

Note – The open source version of the ME Framework is available at: `http://cqme.dev.java.net/framework.html`.

CHAPTER 3

Using the Signature Test Tool

This chapter provides a synopsis of each of the Signature Test tool commands along with their available options and arguments. It contains these sections:

Note – Appendix A includes examples of each command.

Signature Test Tool Basics

The Signature Test tool operates from the command line to generate or manipulate signature files. A signature file is a text representation of the set of public and protected features provided by an API. Test suite developers include it in a finished test suite as a signature reference for comparison to the technology implementation under test. The following list shows the commands that are available.

- `Setup` - Creates a signature file from either an API defined by a specification or a reference API implementation.
- `SignatureTest` - Compares the reference API represented in the signature file to the API under test and produces a report. This is the test that becomes part of a finished test suite.
- `SetupAndTest` - Executes the `Setup` and `SignatureTest` commands in one operation.

- `Merge` - Creates a combined signature file from several signature files representing different Java APIs in one Java runtime environment according to the JSR 68 rules.

The Signature Test tool distribution includes a Java Archive (JAR) file used for developing a signature test and one for distribution within a finished test suite to run its signature test. The description of each follows:

- `sigtestdev.jar` - Contains classes for running the commands used during signature test development.
- `sigtest.jar` - Contains only the classes for running the `SignatureTest` command. This file is distributed in a finished test suite.

Test suite developers perform these operations while using `sigtestdev.jar` to develop a signature test.

1. Run the `Setup` command to create a signature file from either an API defined by a specification or a reference API implementation.
2. Include the files required to run the signature test in the finished test suite distribution.

Reflection and Static Run Modes

Two run modes are available during command execution. These modes determine how the class descriptions are examined and retrieved, as follows:

- **Reflection Mode -** Uses reflection to examine API classes and retrieve information about them. The reflection mode is of greatest advantage when the API to be analyzed has no external class files.
- **Static Mode -** Specified with the `-static` flag, the tool parses only the class files listed in the `-classpath` command-line argument.

Note – In static mode you can test specified classes in another runtime environment. For example, this can be useful to analyze APIs that are part of a Java SE platform 1.4.2 environment when the `SignatureTest` command is run on a Java SE platform version 5.0.

Constant Checking in Differing Run Modes

The requirements related to constant checking differ in binary and source compatibility testing. Although constant checking can be applied to binary compatibility, it is a necessary prerequisite for source code compatibility. Use the `-static` mode to enforce strict constant checking in source code compatibility testing.

When running a signature test in source compatibility mode and using the static mode, constant checking is strict and two way. This means that all the constant fields and their related values specified in the reference API must have the same values in the API under test. Likewise, all the constant fields and their related values specified in the API under test must have the same values in the reference API.

In binary compatibility mode, the requirements related to constant checking are less strict. The signature test verifies that all the constant fields and associated values contained in the reference API are also available in the API under test. If any field values are missing or different, it reports an error. However, the signature test does not report an error if constant values are found in the API under test that are not available in the reference API.

Generics Checking in Binary Mode

The information related to generics is not used by the Java Virtual Machine[1] at runtime. This information is used only by the compiler at compile time.

In binary mode the `SignatureTest` command compares the signatures of parameterized types after omitting the type parameters and arguments from both the signature file and the analyzed API (termed type erasure). This is to ensure that they are compatible at runtime. See *The Java Language Specification, Third Edition,* for a detailed description of type erasure.

The bridge methods that are generated by the compiler during type erasure are not a part of the API and so they are ignored by the Signature Test tool.

`CLASSPATH` and `-classpath` Settings

TABLE 3-1 lists the requirements for setting the `CLASSPATH` environment variable and the `-classpath` argument when running either the `Setup` or `SignatureTest` commands. The table uses the following terms to describe the classes that must be included:

- **Required classes -** All superclasses and superinterfaces of the classes under test

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform

- **Classes under test -** The set of classes specified by a combination of the following options:
  - `-package`
  - `-PackageWithoutSubpackages`
  - `-exclude`

Note – The `Setup` command can only be run in static mode.

TABLE 3-1 Settings for the `Setup` and `SignatureTest` Commands

Variable or Argument	In Reflection Mode	In Static Mode
`CLASSPATH` environment variable	Required. Must contain the following: • `sigtestdev.jar` or `sigtest.jar` • Classes under test • Other required classes, except for bootstrap and extension classes described in "Bootstrap and Extension Classes" on page 16	Required. Must contain either `sigtestdev.jar` or `sigtest.jar`.
`-classpath` argument	Required. Must contain the classes under test.	Required. Must contain the following: • Classes under test • All required classes

Bootstrap and Extension Classes

Bootstrap and extension classes are those located in `rt.jar` and several other JAR files under the *Java-Home*`/jre/lib/` directory, where *Java-Home* is the base directory of the Java platform runtime installation. For example, classes from the `java` package such as `java.lang.Object` are bootstrap classes. Their location is already available to the Java Virtual Machine environment. Because of this, they do not need to be specified in the `CLASSPATH` environment variable when reflection mode is used. Furthermore, bootstrap and extension classes are always loaded from JAR files located in the *Java-Home*`/jre/lib/` directory, regardless of whether they were specified in the class path or not. This is an important feature of the reflection mode.

Signature File `Merge` Rules

The `-Files` argument of the `SignatureTest` command accepts values to specify one or more signature files that are combined to represent an API configuration that is used as input for testing. This resulting API-set can also be combined into a single signature file for testing purposes.

By default the API combination is assumed to be constructed according to the JSR 68 rules. This can be overridden during a Signature Test tool test run by specifying the `-NoMerge` option to run the signature test. The `-NoMerge` option forces the Signature Test tool to use the first class description it encounters if more than one class with the same name is found in the set of signature files specified by the `-Files` argument. In this case it uses the class from the left-most signature file that is specified with `-Files`.

JSR 68-Based Merge

The Java ME platform architecture specified by JSR 68[2] allows for the inclusion of several Java platform APIs in one conforming Java platform runtime environment. The condition for combining these APIs is that any application written for the resulting runtime environment must execute successfully on the combination.

If such a combination exists, it is defined on the basis of the combined sets of APIs with semantics, and the semantics must be compatible with those of all the included components.

The only means of verifying the semantics of the combination is to run the applicable test suite for each API. However, it is possible to automate the creation of a combined set of API signatures, and it is also possible to detect when a combined set cannot be built.

The `Merge` command functionality combines (merges) several input signature files into one resulting signature file, as follows: If A, B, and C are signature files, then A + B yields signature file C, and each of the three signature files represent the corresponding classes of their respective APIs. The `Merge` process constructs the resulting API C out of the two input APIs A and B.

2. http://jcp.org/en/jsr/detail?id=68

`Merge` Command Operative Principles

The `Merge` command operates according to the following principles, where A and B are input APIs that are combined into the resulting API C:

- The `Merge` operation is commutative, so with API A and B, A + B = B + A.
- It recognizes either binary or source compatibility when merging APIs.
- For any application X that is compatible with either API A or B, when A and B are merged then X must be compatible with the resulting API C.
- The resulting API C cannot contain a class that is not found in either of the A or B input APIs. This means that any class in C has to have corresponding classes in either A or B or both A and B.
- API C must not contain a class member that is not found in its corresponding classes in A and B. This applies only to declared class members and not inherited members.
- If some class in A or B, or both, has a member that overrides a member from a superclass, then the corresponding class in C must also have this overriding member.
- Each API element in C has a set of attributes derived from the attributes of its corresponding elements in A and B, and this is the smallest possible set of attributes that does not break compatibility. So if `attr` is an attribute of an element from API C, then `attr` must be defined for the corresponding element from A or B, and `attr` cannot be omitted without breaking compatibility between A and C or between B and C.
- No unnecessary APIs or relationships between classes or interfaces can be introduced.

The basic algorithmic rules for combining two input APIs A and B into a signature file that represents the resulting API C are as follows:

- If one of the input APIs A or B contains an element that the other does not, then this element goes into the resulting signature file of API C without modification except for the following case: If the element in question is the first declared class member in the inheritance chain of input API A or B, and the other input API inherits the same element, then this element represented the resulting API C.
- If both of the input APIs contain two identical elements, only one of them is represented in the resulting API.
- If both of the input APIs contain a corresponding element, but with a different set of attributes, then either of the following occurs:
  - A conflict wherein the resulting API cannot exist.
  - A compromise wherein the new element with a composite set of attributes is created and it is represented in the resulting API-set.

Element Handling by `Merge`

General rules for handling elements of all kinds during the `Merge` process are as follows.

- When there are two different access modifiers select the more visible one.

  For example, if A is a `public int foo`, and B is `protected int foo`, then the merge into C results into `public int foo`.

- If the elements differ in the `final` modifier, do not include it If a class is `final`, then all of its methods are implicitly `final` according to Section 8.4.3.3 of *The Java Language Specification, 2nd Edition*.
- If corresponding elements differ in the `static` modifier, then declare a conflict.

Element-specific rules are as follows:

- If corresponding classes differ in the `abstract` modifier, then declare a conflict.
- Apply the following rules for classes or interfaces and nested classes or interfaces, where for the purpose of this description, c1 and c2 are corresponding classes from the input APIs:

  If a superclass of c1 is a subclass of a superclass of c2, use the superclass of c1 as the superclass for the new element. Otherwise, if a superclass of c2 is a subclass of a superclass of c1, use the superclass of c2 as the superclass for the new element. If neither of the previous two conditions are possible, then declare a conflict.

- For classes or interfaces and nested classes or interfaces, create a combined set of superinterfaces of the corresponding classes and dismiss duplicates. Use the combined set for the new element.
- For methods and constructors, construct a `throws` list as follows:
  - In binary compatibility mode, an empty `throws` list results independently of the source lists.
  - In source compatibility mode, both `throws` lists are normalized as described in TABLE 3-3 before they are compared. If the normalized lists are equal, one is used as the result, otherwise, a conflict is declared.
- Methods that differ in the `abstract` modifier are not included.
- If a method result type is not identical a conflict is declared.
- If a field value type is not identical a conflict is declared.
- If a field element differs in the `volatile` modifier, it is included.
- Process `constant` field values as follows:
  - If one of the fields has a constant value and other does not, include the constant value in the result field.
  - If both fields have a constant value then declare a conflict if the values are different, otherwise include the value in the result field.

`Setup` Command

The `Setup` command has the following synopsis:

`java -jar sigtestdev.jar Setup` [*arguments*]

or

`java com.sun.tdk.signaturetest.Setup` [*arguments*]

TABLE 3-2 describes the available command arguments and the values that they accept. Before running the command, also see these sections: "`CLASSPATH` and `-classpath` Settings" on page 15 and "Case Sensitivity of Command Arguments" on page 22.

Command Description

The `Setup` command accepts a reference implementation of an API as input. The command processes the API input to generate a signature file that represents the API to be used as a reference of comparison for the purpose of signature testing.

`Setup` processes the API input in the static mode by parsing the set of classes specified with the `-classpath` arguments.

Also see Appendix A for an example of the command that you can run.

TABLE 3-2 `Setup` Command Arguments

`Setup` **Option**	**Description**
`-help`	Optional. Displays usage information for available command arguments and exits.
`-debug`	Optional. Enables printing of the stack trace for debugging purposes if `Setup` fails.
`-classpath` *path*	Required. Specifies the path to one or more APIs that generate the signature file. Can contain multiple directories or ZIP or JAR files. The `-package` argument further refines the set of classes specified in `-classpath` (see "`CLASSPATH` and `-classpath` Settings" on page 15). There is no default `-classpath`. Use the path separator appropriate for the platform (identified by `java.io.File.pathSeparator`).

TABLE 3-2 `Setup` Command Arguments *(Continued)*

`Setup` **Option**	**Description**
`-TestURL` *path*	Optional. Specifies the directory location in which to create the signature file as a `file` protocol URL: `file://`*path* Must end in a trailing slash on a UNIX system or a backslash on a Microsoft Windows or DOS system. `Setup` does not support the HTTP protocol.
`-FileName` *file_name*	Required. Specifies the name of the signature file to be created.
`-ClosedFile`	Optional. The default if not specified. Specifies to include in the signature file all direct and indirect superclasses for all required classes (tested classes), even if these superclasses are non-public or from untested packages.
`-NonClosedFile`	Optional. Declines the default `-ClosedFile` mode previously described. Does not include all direct and indirect superclasses and superinterfaces of tested classes in the signature file
`-package` *package_or_class_name*	Optional. Specifies a class or package to be included in the signature file, including its subpackages if a package is specified. The `-package` value acts as a filter on the set of classes specified in `-classpath`. The default is all classes. Repeat the argument to specify multiple entries.
`-PackageWithoutSubpackages` *package*	Optional. Similar to the `-package` option, this specifies a package to be included but without its subpackages. Repeat the option to specify multiple entries.
`-exclude` *package_or_class_name*	Optional. Specifies a package or class to be excluded from the signature file, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the `-package` or the `-PackageWithoutSubpackages` option.
`-verbose`	Optional. Enables error diagnostics for inherited class members.
`-apiVersion` *version_string*	Optional. Specifies the API version string to be recorded in the second line of the signature file, as described in "Signature File Contents" on page 23.
`-static`	Deprecated. The `Setup` command now runs only in static mode.

Case Sensitivity of Command Arguments

The specification of each argument flag at the command line is not case sensitive, but the input value entered immediately after the argument flag is case sensitive.

The following two command lines produce identical results for the `-FileName` flag:

```
% java -jar sigtestdev.jar Setup -FileName name.sig
% java -jar sigtestdev.jar Setup -filename name.sig
```

However, these two might not produce identical results if the host operating system is case sensitive to the file name values entered:

```
% java -jar sigtestdev.jar Setup -FileName name.sig
% java -jar sigtestdev.jar Setup -FileName NAME.sig
```

Signature File Formats

The Signature Test tool has changed signature file formats through progressive versions. TABLE 3-3 lists the existing signature file formats and describes how each relates to a specific Signature Test tool version. In Signature Test tool 2.1, the `SignatureTest` and `Merge` commands read v2.1 and later signature files, and output only v4.0. The v4.0 file format supports added functionality, such as generics and annotations.

TABLE 3-3 Signature File Format Compatibility

Format	Description
`v0`	Generates a signature file with simple class member names. This was the default format in Signature Test tool 1.0, but is not supported by `SignatureTest` command in Signature Test tool 1.3 and later.
`v1`	Generates a signature file with fully qualified class member names. This was the default format in Signature Test tool 1.1. This format includes *non-normalized* exception throw lists for constructors and methods. Normalizing the throw list involves removing all superfluous exception classes. A class is superfluous if it is a subclass of either the `java.lang.RuntimeException` class, or the `java.lang.Error` class, or another class from the same list. This format is not supported by `SignatureTest` command in Signature Test tool 1.3 and later.
`v2`	This is the default format for Signature Test tool 1.2. Generates a signature file with fully qualified class member names and modified `supr` statements. This format includes *normalized* exception throw lists for constructors and methods. This format is not supported by `SignatureTest` command in Signature Test tool 1.3 and later.

TABLE 3-3 Signature File Format Compatibility *(Continued)*

Format	Description
v2.1	This version extends the v2 format to indicate whether an interface is inherited directly or indirectly. It is read by `SignatureTest` command in Signature Test tool 1.3 and later.
`v3.1`	Generates data for JDK software version 5.0 such as generics, annotations, and enums.
`v4.0`	Inherited members are not written to the signature file. Private and default visibility fields and nested classes that can potentially hide visible API elements are tracked. In Signature Test tool 2.1, all output files are of this version.

Also see "Source and Binary Compatibility Modes" on page 9.

Signature File Contents

`Setup` generates each signature file with a mandatory header in the first two lines, followed by the body of the signature file.

Note – Comment lines start with the pound (#) character and can be inserted anywhere after the first two mandatory header lines.

Signature File Header

`Setup` generates the first two mandatory header lines of each signature file as follows:

`#Signature file` *format*

`#Version` *version-string*

With the following variable replacement values:

- *format* is either one of the values described in TABLE 3-3, or empty, that indicates v0.
- *version-string* is a value taken directly from the argument given at the startup command line to the `-apiVersion` option (see TABLE 3-2).

Signature File Body

The remaining body of a signature file immediately follows the header. It contains the following information, which is further clarified in TABLE 3-4:

- For each `public` or `protected` class, all modifiers except `strictfp`, and the fully qualified name of any superclass or interfaces implemented, generic type parameters, and annotations.
- For each `public` or `protected` interface, all modifiers except `strictfp`, and the fully qualified name of any superinterfaces implemented, generic type parameters, and annotations.
- For each `public` or `protected` field, all modifiers except `transient`, the fully qualified name of the field's type and its fully qualified name. If the field is a primitive or string constant, the value of the field is included.
- For each `public` or `protected` method, all modifiers (except `native`, `synchronized`, and `strictfp`), the fully qualified name of the type of returned value, the method's fully qualified name, types of all parameters, and the names of exceptions declared in a `throws` clause.
- For each `public` or `protected` constructor, all modifiers, the fully qualified name of the constructor, types of all parameters, and any exceptions declared in a `throws` clause.

Note – All `private` `types` that are used in the definition of a `public` or `protected` item are substituted by their `public` or `protected` equivalent if possible, otherwise an error is generated. All types included in a signature file are either `public` or `protected` and not `private` or `package` `local`.

TABLE 3-4 further summarizes the contents of a generated signature file. A plus (+) indicates a class modifier is included in a generated signature file and a minus (-) indicates it is ignored for that particular element. A blank cell indicates that the condition does not apply to a cell, for example, a class does not have a transient modifier so it is blank.

TABLE 3-4 Signature File Content Summary

Modifier	Class or Interface	Field	Method	Constructor	Nested Class or Interface
`public`	+	+	+	+	+
`protected`		+	+	+	+
`abstract`	+		+		+
`static`		+	+		+
`final`	+	+	+		+

TABLE 3-4 Signature File Content Summary *(Continued)*

Modifier	Class or Interface	Field	Method	Constructor	Nested Class or Interface
strictfp	–		–		–
transient		–			
volatile		+			
synchronized			–		
native			–		

Using the Setup Command From an Ant Script

The Signature Test tool `Setup` command can be invoked from Ant script. The `sigtestdev.jar` file contains the corresponding Ant task.

Use the following task definition in your Ant build script:

```
<taskdef name="setup"  classname=
"com.sun.tdk.signaturetest.ant.ASetup"  classpath="sigtestdev.jar"/>
```

The table below lists required and optional attributes and nested elements for task "setup":

TABLE 3-5 Setup Attributes Available for Ant Scripts

Attribute	Description
Required	
"package" attribute or nested "package" elements	Corresponds to the `-package` option
"classpath" attribute or nested "classpath" elements	Corresponds to the `-classpath` option
"filename" attribute	Corresponds to the `-filename` option
Optional:	
"fail on error" attribute	Stop the build process if the command exits with an error. Default value is `false`
"apiVersion" attribute	Corresponds to `-apiVersion`. Set API version for signature file
"exclude" attribute or nested "exclude" elements	Corresponds to -exclude option. Specifies package(s) or class(es), which is not required to be tested

The code below provides a sample of using setup command with Ant script:

```
<target name="td" description="Setup task definition">
  <taskdef name="setup" classname="com.sun.tdk.signaturetest.ant.ASetup"
    classpath="${sigtest.home}/sigtestdev.jar"/>
</target>

<target name="setup"
  description="Runs signature test setup for com.acme.openapi package with
subpackages"
  depends="td">
  <setup package="com.acme.openapi" failonerror="true" apiVersion="openapi-v2"
    filename="acme-openapi-v1.sig">
    <classpath>
      <pathelement location="${java.home}/jre/lib/rt.jar"/>
      <pathelement location="${testd.home}/lib/acmeAPIv2.jar"/>
    </classpath>
    <exclude class="com.acme.openapi.NotTested"/>
  </setup>
</target>
```

`SignatureTest` Command

The `SignatureTest` command has the following synopsis:

`java -jar sigtest.jar Test` [*arguments*]

or

`java com.sun.tdk.signaturetest.SignatureTest` [*arguments*]

The second alternative assumes that `sigtest.jar` is on the class path. The rules described in "Case Sensitivity of Command Arguments" on page 22 apply. TABLE 3-6 lists the available arguments.

Note – `SignatureTest` command settings for the `CLASSPATH` environment variable and the `-classpath` argument are the same as those listed for the Setup command in TABLE 3-1.

Command Description

The `SignatureTest` command compares the reference API represented in a signature file to the API under test and produces a report. Depending on the command-line arguments specified, it uses either the reflection or static mode. If the `-classpath` argument is specified, the `SignatureTest` command checks if any extra classes are contained in the APIs it specifies.

See "Signature File Formats" on page 22 for details about supported versions of signature file formats.

TABLE 3-6 `SignatureTest` Command Arguments

Option	Description
`-help`	Optional. Displays usage information for available command arguments and exits.
`-debug`	Optional. Enables printing of the stack trace for debugging purposes if `SignatureTest` fails.
`-static`	Optional. Specifies to run in static mode without using reflection and reports on only the class files specified in the `-classpath` option.

TABLE 3-6 `SignatureTest` Command Arguments *(Continued)*

Option	Description
`-mode [bin | src]`	Optional. Specifies the compatibility mode to use during the signature test, either binary or source, respectively. Defaults to `src`. "Source and Binary Compatibility Modes" on page 9 describes each mode.
`-CheckValue`	Specifies to check the values of primitive and string constants. This option generates an error if a signature file does not contain the data necessary for constant checking.
`-NoCheckValue`	Specifies not to check the values of primitive and string constants.
`-ClassCacheSize` *size_of_cache*	Optional. Used in static mode only. Default is 1024. Specifies the size of the class cache as a number of classes to be held in memory to reduce execution time. Increasing this value dedicates more memory to this function.
`-classpath` *path*	Optional. Specifies the path to one or more APIs to be tested. Defaults to the classes contained in the signature file under test. Can contain multiple directories or ZIP or JAR files. The `-package` argument filters the set of classes specified in `-classpath` (see "`CLASSPATH` and `-classpath` Settings" on page 15). Uses the path separator appropriate for the platform (identified by `java.io.File.pathSeparator`).
`-TestURL` *URL*	Optional. Specifies the directory location of the signature file as a `file` protocol URL: `file://`*path* Must end in a trailing forward slash on a UNIX system or a backslash on a Microsoft Windows or DOS system.
`-FileName` *file_name*	Required if `-Files` is not specified. Specifies the name of a signature file to be used.
`-Files` *file_names*	Required if -FileName is not specified. Use this argument for testing a combination of APIs represented by corresponding signature files. Specifies the names of the signature files to be used delimited by a file separator. The file separator on UNIX systems is a colon (:) character, and on Microsoft Windows systems it is a semicolon (;). See "Signature File `Merge` Rules" on page 17 for details about the rules used for merging.

TABLE 3-6 `SignatureTest` Command Arguments *(Continued)*

Option	Description
`-NoMerge`	Optional. Forces using the first encountered class description if more than one class with the same name is found in the input set of signature files specified by `-Files` option. In this case it uses the class from the left-most signature file that is specified with `-Files`. This option prevents the test from using the default merging behavior according to the JSR 68 rules. See "Signature File `Merge` Rules" on page 17 for details about the rules used for merging.
`-package` *package_or_class_name*	Optional. Specifies a class or package to be reported on, including its subpackages if a package is specified. The default is all classes and packages in the signature file. This argument acts as a filter on the set of classes or packages optionally specified in `-classpath`. Repeat the argument to specify multiple entries.
`-PackageWithoutSubpackages` *package*	Optional. Similar to the `-package` option, specifies a package to be reported on without its subpackages. Repeat the option to specify multiple entries.
`-exclude` *package_or_class_name*	Optional. A package or class to be excluded from the report, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the `-package` or the `-PackageWithoutSubpackages` option.
`-out` *file_name*	Optional. Prints report messages to a specified file instead of standard output.
`-FormatPlain`	Optional. Specifies not to sort report messages, but output them immediately. Allows a decrease in memory consumption compared to the default sorted format of message reporting.
`-FormatHuman` `-H`	Optional. Specifies the human-readable report output format. Simple API changes are presented not as pair of errors ("missing element" and "added element"), but as a single API change.
`-Backward` `-B`	Optional. Specifies migration compatibility check mode.

TABLE 3-6 `SignatureTest` Command Arguments *(Continued)*

Option	Description
`-apiVersion` *version_string*	Optional. Specifies the API version number of the implementation under test to be recorded in the report.
`-verbose`	Optional. Prints all error messages for inherited class members. Default is to remove all these error messages.
`-ErrorAll`	Optional. Specifies to make the signature test more strict by upgrading certain warnings to errors.

Report Formats

You can cause `SignatureTest` command report messages to be sorted (default), or unsorted by specifying the `-FormatPlain` flag at the command line. See CODE EXAMPLE A-4 under "Using the `SetupAndTest` Command" on page 57 to see a sorted report that was generated into a plain text file with the `-out` option. Report messages contain the following types of information:

- The versions of both the reference API and the API under test
- The total number of errors found
- The differences such as added or missing classes, superclasses, fields, or methods
- The fully qualified name of the enclosing class related to any missing or added description
- Two errors for any modified item, one for a missing item and another for an added item
- A description of any incompatibility in the API implementation under test

Sorted Report

Report messages are sorted by default. Unlike the unsorted format, duplicate messages are removed. To accurately compare the error totals between formats, the sorted report prints two error counts: the total number of errors and the total number of duplicates.

The sorted report groups error messages by category of difference with classes within each category ordered alphabetically by name, and empty categories are ignored. This is the category ordering sequence:

- Missing Classes

- Missing Class Descriptions (Modified classes and nested classes)
- Missing Superclasses or Superinterfaces
- Missing Fields
- Missing Constructors
- Missing Methods
- Added Classes
- Added Class Descriptions (Modified classes and nested classes)
- Added Superclasses or Superinterfaces
- Added Fields
- Added Constructors
- Added Methods
- Linkage Errors

Linkage errors occur during testing if the API implementation under test is corrupted, for example, if a superclass or superinterface of a class under test cannot be loaded.

See the sorted report in CODE EXAMPLE A-4.

Unsorted Report

The `-FormatPlain` option specifies an unsorted report. The unsorted format reports messages immediately during execution and duplicate messages are included.

CODE EXAMPLE 3-1 shows an unsorted report that corresponds to the sorted report in CODE EXAMPLE A-4.

CODE EXAMPLE 3-1 Unsorted Report Example

```
Definition required but not found in example.test
    method public int get(int)
Definition found but not permitted in example.test
    method public java.lang.String get(int)
Definition found but not permitted in example.test
    method public void put()
SignatureTest report
Tested version: 2.0
STATUS: Failed. 3 errors
```

Human-Readable Report

The human-readable report format is similar to the sorted report format. Unlike the sorted report format where some changes in the API are presented as pairs of messages (“added element” and “missed element”), the human-readable report format presents these changes as a single message that describes the difference between the APIs. This report recognizes the following cases:

- Modifiers were changed
- Return type was changed
- Parameter types were changed
- Normalized throw list was changed
- Constant value was changed
- Annotation was changed

Unlike the sorted report format, all messages are grouped by class first, rather than message type. The code example below shows the output for an class:

```
Class tests.human.many.E1
  Changed
    method public abstract void tests.human.many.E1.foo() throws java.io.IOException
    - Throws: [java.io.IOException]
    method public abstract void tests.human.many.E1.foo() throws java.io.IOException
    - [public, abstract]
    + [protected]
    field public final static int tests.human.many.E1.i = 0
    - type: int
    + type: char
    field public final static int tests.human.many.E1.i = 0
    - value: 0
```

Using the SignatureTest Command From an Ant Script

The `SignatureTest` command can be invoked from Ant script. Both `sigtestdev.jar` and `sigtest.jar` contain corresponding Ant tasks. Unlike the ordinary `SignatureTest` command, its Ant wrapper can run the test only in static mode—reflection mode is not supported. Use the following task definition in your Ant build script:

```
<taskdef name="test"  classname=
"com.sun.tdk.signaturetest.ant.ATest"  classpath="sigtest.jar"/>
```

TABLE 3-7 lists required and optional attributes and nested elements for task “test”.

TABLE 3-7 SignatureTest Attributes Available for Ant Scripts

Attribute	Description
Required:	
"package" attribute or nested "package" elements	Corresponds to the `-package` option
"classpath" attribute or nested "classpath" elements	Corresponds to the `-classpath` option
"filename" attribute	Corresponds to the `-filename` option
Optional:	
"failonerror" attribute	Stops the build process if the command exits with an error. Default is false
"apiVersion" attribute	Corresponds to the `-apiVersion` option. Sets the API version for signature files.
"exclude" attribute or nested "exclude" elements	Corresponds to the `-exclude` option. Specifies package(s) or class(es), do not require testing.
"binary" attribute	Corresponds to the "`-mode bin`" option. Runs the test in binary mode. Default is "false".
"errorAll" attribute	Corresponds to the `-errorAll` option. Handles warnings as errors. Default is "false".
"debug" attribute	Corresponds to the `-debug` option. Prints debug information such as detailed stack traces. Default is "false".
"backward" attribute	Corresponds to the `-Backward` option. Runs backward compatibility checking. Default is "false".
"formatHuman" attribute	Corresponds to the `-formatHuman` option. Processes human readable error output. Default is "false".
"output" attribute	Corresponds to the `-out` option. Specifies the report file name.

The following code sample shows how to use the SignatureTest command with an Ant script.

```
<target name="td" description="Signature test task definition">
  <taskdef name="test" classname="com.sun.tdk.signaturetest.ant.ATest"
    classpath="${sigtest.home}/sigtest.jar"/>
</target>

<target name="test"
  description="Runs migration compatibility test for com.acme.openapi
  package with subpackages, tests v2 against v1 signature file"
  depends="td">
  <test failonerror="true" apiVersion="openapi-v2"
    filename="acme-openapi-v1.sig" backward="true" output="st_report.txt">
    <package name="com.acme.openapi" />
    <exclude class="com.acme.openapi.NotTested"/>
    <classpath>
      <pathelement location="${java.home}/jre/lib/rt.jar"/>
      <pathelement location="${testd.home}/lib/acmeAPIv2.jar"/>
    </classpath>
 </test>
</target>
```

`SetupAndTest` Command

The `SetupAndTest` command has the following synopsis:

```
java -jar sigtestdev.jar SetupAndTest [arguments]
```

or

```
java com.sun.tdk.signaturetest.SetupAndTest [arguments]
```

The second alternative assumes that `sigtestdev.jar` is on the class path. The rules described in "Case Sensitivity of Command Arguments" on page 22 apply. TABLE 3-8 lists the available arguments.

Command Description

`SetupAndTest` is a wrapper command that runs only in the static mode to execute the `Setup` and `SignatureTest` commands in one operation. It performs these functions:

1. Calls the `Setup` command to create a signature file from the reference API specified as input. It creates a temporary signature file if no name is specified for it with the `-FileName` option.
2. Calls the `SignatureTest` command to make the comparison and produce a report.

TABLE 3-8 `SetupAndTest` Command Argument

`-help`	Optional. Displays usage information for available command arguments and exits.
`-reference` *path*	Required. Specifies the path to the classes to be used for creating the signature file.
`-test` *path*	Required. Path to the packages being tested.
`-FileName` *file*	Optional. Specifies the signature file name. The command execution uses a temporary file if not specified.
`-mode [bin | src]`	Optional. Specifies the compatibility mode to use during the signature test, either binary or source, respectively. Defaults to `src`. "Source and Binary Compatibility Modes" on page 9 describes each mode.

TABLE 3-8 `SetupAndTest` Command Argument *(Continued)*

`-package` *package_or_class_name*	Optional. Specifies a class or package to be reported on, including its subpackages if a package is specified. This argument acts as a filter on the set of classes or packages that are tested and reported on. The default is all classes and packages. Repeat the argument to specify multiple entries.
`-PackageWithoutSubpackages` *name*	Optional. Specifies package to be tested excluding subpackages.
`-exclude` *name*	Optional. A package or class to be excluded from testing, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the `-package` or the `-PackageWithoutSubpackages` option.
`-ClassCacheSize` *size_of_cache*	Optional. Specifies the size of the class cache as a number of classes to be held in memory to reduce execution time. Increasing this value dedicates more memory to this function. Defaults to 100.
`-CheckValue`	Optional. Specifies to check the values of primitive and string constants. This option generates an error if a signature file does not contain the data necessary for constant checking.
`-NoCheckValue`	Optional. Specifies not to check the values of primitive and string constants.
`-verbose`	Optional. Enables error diagnostics for inherited class members.
`-apiVersion` *version_string*	Optional. Specifies the API version number of the implementation under test to be recorded in the generated report.
`-out` *file_name*	Optional. Prints report messages to a specified file instead of standard output.

TABLE 3-8 `SetupAndTest` Command Argument *(Continued)*

`-FormatPlain`	Optional. Specifies not to sort report messages, but output them immediately. Allows a decrease in memory consumption compared to the default sorted format of message reporting.
`-FormatHuman` `-H`	Optional. Specifies the human-readable report output format. Simple API changes are presented not as pair of errors ("missing element" and "added element"), but as a single API change.
`-Backward` `-B`	Optional. Specifies migration compatibility check mode.

`Merge` Command

The `Merge` command has the following synopsis:

```
java -jar sigtest.jar Merge [arguments]
```

or

```
java com.sun.tdk.signaturetest.Merge [arguments]
```

The second alternative assumes that `sigtest.jar` is on the class path. The rules described in "Case Sensitivity of Command Arguments" on page 22 apply. TABLE 3-9 lists the available arguments.

Command Description

The `Merge` command combines (merges) a number of input signature files into one resulting signature file. See "Signature File `Merge` Rules" on page 17 for details about the rules used for merging, and "Signature File Formats" on page 22 for details about supported versions of signature file formats. TABLE 3-9 lists the `Merge` command options.

TABLE 3-9 `Merge` Command Arguments

Argument	Description
`-help`	Optional. Displays usage information for available command arguments and exits.
`-Files`	Required. Specifies the names of the signature files to be merged delimited by a file separator. The file separator on UNIX systems is a colon (:) character, and on Microsoft Windows systems it is a semicolon (;). See "Signature File `Merge` Rules" on page 17 for details about the rules used for merging.
`-Write`	Required. Specifies the resulting output signature file.
`-Binary`	Optional. Specifies to use the binary merge mode. See "Source and Binary Compatibility Modes" on page 9.

PART II API Coverage Tool

Describes how you can use the API Coverage tool to estimate the test coverage a test suite provides for an implementation of a specified API. It does this by determining how many public class members the test suite references within the API specification. The tool uses a signature file representation of the API specification as the source of specification analysis. It does not process a formal specification in any form.

CHAPTER 4

Introduction to API Coverage Tool

The API Coverage tool is used to estimate the test coverage that a test suite under development affords to implementations of its related API specification. It does this by determining how many public class members the test suite references within the API specification that it is designed to test against. The tool uses a signature file representation of the API specification as the source of specification analysis. It does not process a formal specification in any form.

The API Coverage tool code is contained in both the `sigtestdev.jar` and `sigtest.jar` files. Additional installation is not required. See Chapter 5 for details about running API Coverage tool.

Static API Coverage Analysis

The tool operates on the principle that a reference to an API member from within a test class indicates a test of that member by the test suite. The ratio of referenced class members to the total number of class members calculated for an API yields a percentage of test coverage for the API.

This method is called *static* API coverage analysis because it does not actually run any tests from the test suite. Since it does not dynamically determine which API members are actually accessed by the tests, the coverage calculation expresses only an estimated percentage of test coverage.

Major Source of Error

Static analysis cannot correctly predict the outcome of virtual calls to overridden methods that are resolved at runtime through dynamic method dispatch. The frequency of this type of overridden method can vary between differing

implementations of the same API specification. This makes it difficult to formulate an exact percentage of test suite coverage when using static analysis (in spite of the fact that the implementations may all be binary compatible and correctly implement the specification).

Tests that make dynamic calls to API members are not be recognized by API Coverage tool. This my cause some test calls to not be accurately measured.

Advantages of Static Coverage Analysis

Static API coverage testing provides the following advantages over dynamic methods.

- Static testing is more lightweight. Testing is easier to setup and quicker to run than dynamic methods.
- Static testing is easier to automate and provides more consistent results because it is not affected by external conditions such as machine load, or network traffic.
- Static testing is more practical for gathering results for very large APIs. A test suite and its associated API might include many thousands of tests and associated API class members making it very cumbersome to instrument dynamic tests.
- Static testing allows you to quickly *estimate* the quality of test coverage for all APIs, especially APIs that are difficult to test dynamically.

How It Works

The static API coverage algorithms examine precompiled test suite test classes to determine the members that they reference. This includes inner classes and fields as well as constructors, although constructors cannot be inherited.

The algorithms are based on the fact that the constant pool of any class file holds all of its external class references. This constant pool consists of the following related records:

`CONSTANT_Fieldref`

`CONSTANT_Methodref`

`CONSTANT_InterfaceMethodref`

Each of these records contains the fully qualified name of an external class and the name of the class member referenced. For a method, this includes the signature of the method and its return type.

Level of Accuracy During Analysis

TABLE 1 lists example scenarios encountered during static coverage analysis and their related potential for error. The table references the following objects:

- `Q.m` is a method referenced by the test suite.

  Where `Q` is the fully qualified class name, and `m` is the descriptor of the called method (including the name, list of arguments, and return type).
- `SubQ` is a subclass of, and `SupQ` is a superclass of class `Q`.

The main potential for an inaccurate coverage measurement exists when a great many members are overridden in subclasses of an implementation (as described in condition #2 of TABLE 1).

TABLE 1 Example Scenarios and Potential Errors

Condition	Scenario	Result
#1: Object `x` is of type `Q`	Class `Q` declares method `m` and `Q.m` is the method called	`Q.m` is correctly marked as covered
	Method `m` is inherited from superclass `SupQ` and is not declared in `Q`; method `SupQ.m` is called	Correctly marks either `Q.m` or `SupQ.m` as covered, depending on the calculation mode in use (described later in "Coverage Analysis Modes" on page 44)
#2: Object `x` is of type `SubQ`, a subclass of `Q`	No subclass or superclass of `Q` overrides method `m`; `Q.m` is called	Correctly marks `Q.m` as covered
	A subclass of `Q` *does* override method `m` and the overriding method is called; if there are multiple inherited subclasses, exactly which method is called cannot be correctly identified before runtime	`Q.m` is incorrectly marked as covered; *this scenario is the main source of analysis errors*
#3: A method is called by means of reflection	Uses: `Method.invoke(Object, Object[])`	No method is marked as covered, assuming that `java.lang.reflect` is not in the API under test; this case cannot be correctly resolved

Coverage Analysis Modes

The API Coverage tool uses these two modes of analysis:

- **Real World Mode:** Returns a fairly accurate estimate based on input from one specific API implementation, such as a reference implementation. You can then compare the real world results to those of the worst case mode.
- **Worst Case Mode:** Returns an estimate based on a hypothetical API in which every class overrides or hides all members from its superclass. This scenario is highly unlikely to occur in actual practice. You use this mode by extrapolating its results into those of the real world case to estimate the possible range in test coverage that a test suite will provide in the field over a number of differing implementations.

Chapter 5 describes how to set up and use the API Coverage tool.

CHAPTER 5

Using the API Coverage Tool

API Coverage tool requires these environmental settings and components:

- A properly configured certified Java Platform, Standard Edition ("Java SE platform") runtime version 1.4 or later
- One of the following JAR files in your class path:
  - `sigtestdev.jar`
  - `sigtest.jar`

  The API Coverage tool `Main` class is contained in both JAR files.

The API Coverage tool is a command-line utility that executes in a Java application environment. It generates static API test coverage reports in either plain text or HTML format based on the following two input items:

- The test suite being tested for its coverage, represented by a set of test class files
- A signature file that forms a text representation of the public members of an API that must be tested by the test suite

You can use the `Setup` command of the Signature Test tool to create the appropriate signature file for use by the API Coverage tool. The `Setup` command is described in Chapter 3.

Running API Coverage Tool

API Coverage tool command synopsis:

```
% java -jar apicover.jar [arguments]
```

or

```
% java com.sun.tdk.apicover.Main [arguments]
```

The second alternative assumes that `apicover.jar` is on the class path. The rules described in "Case Sensitivity of Command Arguments" on page 22 apply. TABLE 5-1 lists the available arguments.

TABLE 5-1 API Coverage Tool Command Arguments

Argument	Description
`-help`	Optional. Displays usage information for available command arguments and exits.
`-ts` *path*	Required. Cannot be repeated. Path to the test suite classes to be analyzed. Also accepts a JAR file.
`-tsInclude` *package_name*	Optional. Can be repeated. Recursively include classes from the specified test suite package.
`-tsIncludeW` *package_name*	Optional. Can be repeated. Include classes from the specified test suite package without subpackages.
`-tsExclude` *package_name*	Optional. Can be repeated. Recursively exclude classes from the specified test suite package.
`-excludeList` *filename*	Optional. Can be repeated. Specifies the file name of an exclude list which contains API elements to be excluded from the coverage calculation. "Exclude List" on page 49 describes the exclude list format.
`-excludeInterfaces`	Optional. Exclude all interface classes. This argument is superfluous if used with `-excludeAbstractClasses`.
`-excludeAbstractClasses`	Optional. Exclude all abstract classes including interfaces.
`-excludeAbstractMethods`	Optional. Exclude all abstract methods from all classes and interfaces
`-excludeFields`	Optional. Exclude all fields from all classes and interfaces. cannot be used with the `-includeConstantField` argument.

TABLE 5-1 API Coverage Tool Command Arguments *(Continued)*

Argument	Description
`-includeConstantFields`	Optional. Specifies to include all final fields of any type in the coverage calculation. cannot be used with the `-excludeFields` argument. The default, without this option, is to remove from the coverage calculation all final fields of primitive types and type `java.lang.String`. See footnote *.
`-validate [yes | no]`	Optional. Specifies whether to validate test suite classes. Defaults to `yes`.
`-api` *path/filename*	Required. Cannot be repeated. Specifies the location of the signature file representing the API under examination. The *path* defaults to the working directory. Accepts `v1` and `v2` format signature files generated by the Signature Test Tool `Setup` command, and rejects `v0`. See the `-FileFormat` argument description in Chapter 3.
`-apiInclude` *package_name*	Optional. Can be repeated. Recursively include classes from the specified API package.
`-apiIncludeW` *package_name*	Optional. Can be repeated. Include classes from the specified API package without subpackages.
`-apiExclude` *package_name*	Optional. Can be repeated. Recursively excludes classes from the specified API package.
`-mode [w | r]`	Optional. Specifies the mode of coverage calculation as `w` for worst case or `r` for real world. Defaults to worst case.
`-detail [0 | 1 | 2 | 3 | 4]`	Optional. Specifies the level of report detail as an integer from 0–4. Defaults to 2. See TABLE 5-2.
`-format [plain | html]`	Optional. Specifies the report format as plain text or HTML. Defaults to `plain`. See TABLE 5-3.
`-report` *path/filename*	Optional. Where to place the generated report file. The *path* argument defaults to the working directory when only a file name is specified; otherwise, defaults to standard out.
`-out` *path/filename*	Optional. Generates a specially formatted report file for use by other applications, and places it at the specified location/file name. The *path* defaults to the working directory. See "Special Report File" on page 48.

* Constant fields such as final fields of primitive and `String` types are initialized at compile-time with a constant expression. The Java technology-based compiler usually optimizes them and replaces them with their values. In such cases, test suite classes would not contain a reference to the field, even if the field were used/referenced in the test suite. For this reason, the tool removes all constant fields from the calculation by default to make coverage calculations more consistent. The `-includeConstantFields` option provides a means for changing this behavior.

TABLE 5-2 describes the reporting levels resulting from the five settings available with the `-detail` argument.

TABLE 5-2 Report File Contents for Levels of Detail

Detail Setting	Package Summary	Class Summary	Members Not Covered	Covered Members
0	X			
1	X	X		
2	X	X	X	
3	X	X		X
4	X	X	X	X

TABLE 5-3 lists how the reported members are indicated in plain text versus HTML reports.

TABLE 5-3 Member Indicators Used within Each Report Format

Format	Declared Members	Inherited Members	Covered Members	Non-Covered Members
Plain text	Not Indicated	Not Indicated	+	–
HTML	Black text	Blue text	Disc `<LI TYPE="disc">`	Circle `<LI TYPE="circle">`

Special Report File

The special report file specified with the `-out` argument contains the same coverage data that the standard report file contains. The main difference is that this format is easier for other applications to parse.

The special report file is essentially a table with these two columns delimited by a tab character:

- The left column provides the coverage counter value which can be zero.
- The right column provides the fully qualified name of the corresponding API member: a class, constructor, method, or field.

The `-mode` argument affects the special report file, while the `-detail` setting has no effect on it.

Exclude List

In some cases a test suite either cannot or should not test certain elements of an API, whether they be a class, a method, or field. In this case you can use the `-excludeList` *filename* argument (described in TABLE 5-1) to make the coverage calculation more accurate by excluding the elements listed in the specified exclude list file(s).

The exclude list is a text file that specifies each entry for exclusion on a separate line, as follows:

- *Package name* specified to recursively exclude all classes of the package
- *Class name* specified as a fully qualified class name
- *Field name* specified using a fully qualified class name: *classname.fieldname*
- *Method name* specified using a fully qualified class name: *classname.methodname* (*parameters*)

Exclude list processing follows these rules:

- Regards a line starting with the # character as a comment.
- Ignores empty lines.
- Silently ignores an entry if it is not found in the API.

Note – Each entry specified for exclusion should match its appearance in a report file. For example, you must specify an inner class with the `$` character prepended to its name, like this: *Outer$Inner*

PART III Appendix

This appendix contains step-by-step examples that show how to use the Signature Test tool.

APPENDIX **A**

Signature Test Tool Quick Start Examples

This appendix contains the following two step-by-step examples that show you how to use the Signature Test tool:

- Example 1: Compare Two Different Implementations of the Same API
- Example 2: Merge Two Signature Files

Sample sources used in the examples, along with simple UNIX operating system shell and Windows batch scripts are provided in the `examples/usersguide` directory. You can use these to help run the examples. The README file in that directory explains how to use them.

Note – The examples use UNIX operating system syntax.

CODE EXAMPLE A-1 and show the two implementations of the test API used throughout these examples.

CODE EXAMPLE A-1 Version 1.0 of `test.java`

```
package example;
public class test {
    public <T> T get (T x) {
        return x;
    }
}
```

CODE EXAMPLE A-2 Version 2.0 of `test.java`

```
package example;

public class test {

    public String get (int x) {
        return "";
    }

    public void put () {
    }
}
```

Example 1: Compare Two Different Implementations of the Same API

You can compare two implementations of an API in two ways:

- Run the `Setup` command to create a signature file for the first implementation and then run the `SignatureTest` command to compare them. This is shown in the first part of this example: "Using the `Setup` and `SignatureTest` Commands" on page 54.
- Run the `SetupAndTest` command. The command combines the `Setup` and `SignatureTest` commands. This is shown in the second part of this example: "Using the `SetupAndTest` Command" on page 57.

▼ Using the `Setup` and `SignatureTest` Commands

This example shows how to compare the following implementations of the `test.java` API using the `Setup` and `SignatureTest` commands.

1. **Make** *SigTest-directory*`/examples/usersguide` **the current directory.**
2. **Compile the two implementation files.**

   SigTest tool works with class files. Use the following commands to compile the source files into class file:

   ```
   % javac -d V1.0 V1.0/test.java
   ```

```
% javac -d V2.0 V2.0/test.java
```

3. **Set the CLASSPATH environment variable to include the path to `sigtestdev.jar`.**

```
% setenv CLASSPATH=../../lib/sigtestdev.jar
```

4. **Set the JAVA_HOME environment variable to include the base directory of the Java platform runtime installation on your system.**

```
% setenv JAVAHOME=java_runtime_path
```

5. **Use the `Setup` command to create a signature file for the V1.0 implementation.**

   Note that the "\" characters are used to break long lines. Type this as a single line at the command line.

```
% java com.sun.tdk.signaturetest.Setup \
  -classpath V1.0:$JAVA_HOME/jre/lib/rt.jar \
  -static \
  -apiVersion V1.0 \
  -package example \
  -fileName test.sig
```

   The `Setup` command emits console output similar to the following:

```
Constant checking: on
Found in total: 16260 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

6. **Confirm that the contents of the signature file are correct by comparing it to** CODE EXAMPLE A-3**.**

CODE EXAMPLE A-3 The `test.sig` File

```
#Signature file v4.0

CLSS public example.test
cons public test()
meth public <%0 extends java.lang.Object> {%%0} get({%%0})
supr java.lang.Object

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws java.lang.InterruptedException
meth public final void wait(long) throws java.lang.InterruptedException
meth public final void wait(long,int) throws java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()
```

7. **Use the `SignatureTest` command to compare the V2.0 API signature with the V1.0 signature file you created in Step 5.**

   ```
   % java com.sun.tdk.signaturetest.Setup \
     -classpath V2.0:$JAVA_HOME/jre/lib/rt.jar \
     -static \
     -package example \
     -fileName test.sig \
     -out report.txt
   ```

   The `-out` option directs the command to write its results to `report.txt`.

   A message similar to the following is written to the console when the command terminates:

   ```
   See log recorded to file report.txt for more details.
   STATUS:Failed.3 errors
   ```

8. **Confirm that the contents of the `report.txt` file are correct by comparing it to CODE EXAMPLE A-4.**

   Note that there may be some minor differences because of system-specific information.

CODE EXAMPLE A-4 The `report.txt` File

```
SignatureTest report
Tested version:
Check mode: src [throws normalized]
Constant checking: off


Missing Methods
---------------

example.test:       method public <%0 extends java.lang.Object> {%%0}
example.test.get({%%0})

Added Methods
-------------

example.test:       method public java.lang.String example.test.get(int)
example.test:       method public void example.test.put()


STATUS:Failed.3 errors
```

▼ Using the `SetupAndTest` Command

The `SetupAndTest` command combines the functionality of the `Setup` and `SignatureTest` commands into a single command.

1. **Follow steps 1 - 4 in the previous example.**
2. **Use the following `SetupAndTest` command to create a signature file.**

```
% java $CLASSPATH com.sun.tdk.signaturetest.SetupAndTest \
  -reference V1.0:$JAVA_HOME/jre/lib/rt.jar \
  -test V2.0:$JAVA_HOME/jre/lib/rt.jar \
  -package example
```

The `SetupAndTest` command reports results similar to CODE EXAMPLE A-5 to standard output.

CODE EXAMPLE A-5 `SetupAndTest` Command Output Example

```
Invoke Setup ...
Class path: "V1.0;C:\java\jdk1.5.0_06\/jre/lib/rt.jar"
Constant checking: on
Found in total: 12749 classes
```

CODE EXAMPLE A-5 `SetupAndTest` Command Output Example *(Continued)*

```
Selected by -Package: 1 classes
Written to sigfile: 2 classes
Invoke SignatureTest ...
SignatureTest report
Tested version: 2.0
Check mode: src [throws normalized]
Constant checking: on


Missing Methods
---------------

example.test:      method public <%0 extends java.lang.Object> {%%0} example.te
st.get({%%0})

Added Methods
-------------

example.test:       method public java.lang.String example.test.get(int)
example.test:       method public void example.test.put()


STATUS:Failed.3 errors
```

Example 2: Merge Two Signature Files

This example shows how to use the `Merge` command to accomplish the following:

1. Compile three `.java` files to produce `.class` files.
2. Run the `Setup` command on each `.class` file to produce a signature file.
3. Use the `Merge` command to combine the files.

▼ Running Merge Examples

These `Merge` examples use the environment variables listed in TABLE A-1.

TABLE A-1 Environment Variable Settings for Merge Examples

Environment Variable	Description
`CLASSPATH`	Must include the `sigtestdev.jar` file. For example: `/sigtest-1.5/lib/sigtestdev.jar`.
`RT_JAR`	The location of the runtime `rt.jar` file. For example: `/opt/jdk1.5.0_09/jre/lib/rt.jar`.

1. **Set up the environment with the environment variable settings in TABLE A-1.**
2. **Create each of the files shown in CODE EXAMPLE A-6, CODE EXAMPLE A-7, and CODE EXAMPLE A-8. Create each in a different directory but with the same file name.**

   Create directories named `1`, `2`, and `3`. Add the appropriate file to each directory.

CODE EXAMPLE A-6 File `./1/A.java`

```
package x;
public class A {
    public void abc() {}
    public void foo() {}
}
```

CODE EXAMPLE A-7 File `./2/A.java`

```
package x;
public class A {
    public void abc() {}
    public void bar() {}
}
```

CODE EXAMPLE A-8 File `./3/A.java`

```
package x;
public class A {
    public static void abc() {}
}
```

3. **Run these commands to compile each file into a separate `x` subdirectory:**

```
% javac -d 1 1/A.java
% javac -d 2 2/A.java
% javac -d 3 3/A.java
```

This creates the following class files:

- `./1/x/A.class`
- `./2/x/A.class`
- `./3/x/A.class`

4. **Run the following three `Setup` commands on each `A.class` file to produce the three `x#.sig` files like the ones shown in** CODE EXAMPLE A-9**,** CODE EXAMPLE A-10 **and** CODE EXAMPLE A-11**.**

   a. **Command #1:**

```
% java -cp 1:$CLASSPATH com.sun.tdk.signaturetest.Setup \
  -static -classpath 1:$RT_JAR -package x -FileName x1.sig
```

The command generates the `./x1.sig` file like the one shown in CODE EXAMPLE A-9 and produces a similar console message.

```
Class path: "1:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

b. Command #2:

```
% java -cp 2:$CLASSPATH com.sun.tdk.signaturetest.Setup \
  -static -classpath 2:$RT_JAR -package x -FileName x2.sig
```

The command generates the `./x2.sig` file shown in CODE EXAMPLE A-10 and produces a console message similar to this indicating successful setup.

```
Class path: "2:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

c. Command #3:

```
% java -cp 3:$CLASSPATH com.sun.tdk.signaturetest.Setup \
  -static -classpath 3:$RT_JAR -package x -FileName x3.sig
```

The command generates the `./x3.sig` file shown in CODE EXAMPLE A-11 and produces a console message similar to this indicating successful setup.

```
Class path: "3:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.
```

5. **Run the following command to merge** `x1.sig` **and** `x2.sig` **and produce the** `x1+x2.sig` **file shown in** CODE EXAMPLE A-12**:**

```
% java -cp $CLASSPATH com.sun.tdk.signaturetest.Merge -Files \
  x1.sig:x2.sig -Write x1+x2.sig
```

The command generates a console message similar to the following message:

```
Warning: class java.lang.Throwable not found
STATUS:Passed.
```

6. **Run this command to merge** `x2.sig` **and** `x3.sig` **attempting to produce the** `x2+x3.sig` **file:**

```
% java -cp $CLASSPATH com.sun.tdk.signaturetest.Merge -Files
  x2.sig:x3.sig -Write x2+x3.sig
```

The command prints a message to the console similar to the following. it indicates a conflicting `static` modifier, and no signature file is created:

```
x.A.abc : <static> modifier conflict
STATUS:Error.Error
```

The conflict occurs because `x2.sig` contains this method (see CODE EXAMPLE A-10):

```
meth public void x.A.abc()
and x3.sig contains this method:
meth public static void x.A.abc()
```

Example Result Files

This section contains the files that are generated from the previous `Merge` examples.

CODE EXAMPLE A-9 Contents of `./x1.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()
CLSS public x.A
cons public A()
meth public void abc()
meth public void foo()
supr java.lang.Object
```

CODE EXAMPLE A-10 Contents of `./x2.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public A()
meth public void abc()
meth public void bar()
supr java.lang.Object
```

CODE EXAMPLE A-11 Contents of `./x3.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
```

CODE EXAMPLE A-11 Contents of `./x3.sig` *(Continued)*

```
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public A()
meth public static void abc()
supr java.lang.Object
```

CODE EXAMPLE A-12 Contents of `x1+x2.sig`

```
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public Object()
meth protected java.lang.Object clone() throws
java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws
java.lang.InterruptedException
meth public final void wait(long) throws
java.lang.InterruptedException
meth public final void wait(long,int) throws
java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public A()
meth public void abc()
meth public void bar()
meth public void foo()
supr java.lang.Object
```

APPENDIX B

API Coverage Tool Quick Start Examples

This section contains step-by-step examples that show how to produce test coverage reports using the API Coverage tool. The examples are located in the following directory:

SIGTEST_HOME`/examples/userguide/apicover`

The appendix consists of the following examples. The examples build upon each other.

- Set Up the API Coverage Tool Ant Build Script
- Build API Coverage Tool Examples
- Run the Example Test Suite
- Generate a Signature File for the Tested API
- Use API Coverage Tool to Calculate Test Coverage

Note – The examples use UNIX operating system syntax.

▼ Set Up the API Coverage Tool Ant Build Script

1. **Make** *SIGTEST_HOME*`/examples/userguide/apicover` **the current directory.**
2. **Make sure that the** `JAVA_HOME` **environment variable refers to JDK 5.0 or higher.**
3. **Edit the following file:**

SIGTEST_HOME`/examples/userguide/apicover/build.properties`

Set the `junit.jar` property to point to the `junit.jar` file. Obtain this file from version 3.7 or higher of the JUnit test harness product.

4. **Make sure Apache Ant 1.7 or higher is available and properly configured on your system.**

▼ Build API Coverage Tool Examples

1. **Run the `ant` command and specify the "build" target.**

   `%ant build`

   This command builds the following:

   - API classes
   - Example application (`helloWorld.jar`)
   - Test suite
   - API documentation (`doc/` directory)

2. **Run the example application to verify that it built correctly.**

   `%java -jar helloWorld.jar`

   The application's output should look like this:

```
Good morning, Brazil!
Good morning, USA!
Good evening, China!
Good afternoon, Czech Republic!
Good afternoon, Russia!
Good afternoon, United Kingdom!
Good evening, India!
Good afternoon, Mali!
Hello, world!
```

▼ Run the Example Test Suite

1. **Use the Ant "test" target to run the test suite.**

   `%ant test`

   The test suite runs and produces the following output:

```
[junit] Running TestGreet
[junit] Tests run: 4, Failures: 0, Errors: 1, Time elapsed: 0.219 sec
[junit] Test TestGreet FAILED
```

▼ Generate a Signature File for the Tested API

1. **Use the Ant "sigtest" target to generate the signature file.**

   `%ant sigtest`

   The "sigtest" target executes the following command:

   ```
   java -jar sigtestdev.jar setup -filename reports/greetapi.sig
   -package com.sun.tdk.samples.helloworld.api -classpath
   helloWorld.jar:$JAVA_HOME/jre/lib/rt.jar
   ```

   This command generates the file `reports/greetapi.sig`. While the command runs you should see the following output:

```
sigtest:
     [java] Class path: "helloWorld.jar:/opt/java/jdk1.5.0_16/jre/lib/rt.jar"
     [java] Constant checking: on
     [java] Found in total: 13217 classes
     [java] Selected by -Package: 5 classes
     [java] Written to sigfile: 8 classes
     [java] STATUS:Passed.
     [java] Java Result: 95
```

▼ Use API Coverage Tool to Calculate Test Coverage

1. **Use the Ant "apicov" target to calculate static test coverage**

   `%ant apicov`

   Test coverage is calculated using three different modes. These modes are described in the following sections.

Worst Case Mode

The script executes the following command:

```
java -jar apicover.jar -ts test_classes -api reports/greetapi.sig
-apiinclude com.sun.tdk.samples.helloworld.api -mode w -report
reports/cov-worst-case.txt
```

This mode reports that about of 37% of the API is covered by the tests. Some members that are defined in superclasses, such as `finalize()` or `clone()`, are marked as not covered. Some fields are also marked as not covered by the tests. The output written to a file named `reports/cov-worst-case.txt` and should appear as shown below.

```
Coverage Report

================================================================================
Package                                  classes
    Class                                          members tested  %%
        Member
================================================================================
com                                      4         35      13      37%
--------------------------------------------------------------------------------
com.sun                                  4         35      13      37%
--------------------------------------------------------------------------------
com.sun.tdk                              4         35      13      37%
--------------------------------------------------------------------------------
com.sun.tdk.samples                      4         35      13      37%
--------------------------------------------------------------------------------
com.sun.tdk.samples.helloworld           4         35      13      37%
--------------------------------------------------------------------------------
com.sun.tdk.samples.helloworld.api       4         35      13      37%

    Greet                                          3       3       100%

    GreetFactory                                   6       1       16%
      - clone()
      - equals(java.lang.Object)
      - finalize()
      - hashCode()
      - toString()

    Places                                         16      5       31%
      - finalize()
      - getCountry()
      - toString()
      - valueOf(java.lang.Class,java.lang.String)
      - valueOf(java.lang.String)
      - Brazil
      - China
      - Czech
      - India
```

```
      - Mali
      - UK

    TimeOfDay                                        10      4       40%
      - finalize()
      - valueOf(java.lang.Class,java.lang.String)
      - AFTERNOON
      - EVENING
      - MORNING
      - NIGHT
================================================================================
Overall coverage                          4         35      13      37%
================================================================================

Configuration
ts   test_classes
api  reports/greetapi.sig
mode w
```

Real World Mode

The script executes the following command:

```
java -jar apicover.jar -ts test_classes -api reports/greetapi.sig
-apiinclude com.sun.tdk.samples.helloworld.api -mode r -report
reports/cov-realworld.txt
```

This estimation does not take into account overridden members defined in superclasses. Overall coverage is 52%.

The report is named `reports/cov-realworld.txt` and is shown below.

```
Coverage Report

================================================================================
Package                                   classes
    Class                                           members tested  %%
        Member
================================================================================
com                                       4         25      13      52%
--------------------------------------------------------------------------------
com.sun                                   4         25      13      52%
--------------------------------------------------------------------------------
com.sun.tdk                               4         25      13      52%
--------------------------------------------------------------------------------
com.sun.tdk.samples                       4         25      13      52%
--------------------------------------------------------------------------------
```

```
com.sun.tdk.samples.helloworld          4         25      13      52%
------------------------------------------------------------------------------
com.sun.tdk.samples.helloworld.api      4         25      13      52%

    Greet                                         3       3       100%

    GreetFactory                                  1       1       100%

    Places                                        13      5       38%
      - getCountry()
      - valueOf(java.lang.String)
      - Brazil
      - China
      - Czech
      - India
      - Mali
      - UK

    TimeOfDay                                     8       4       50%
      - AFTERNOON
      - EVENING
      - MORNING
      - NIGHT
==============================================================================
Overall coverage                        4         25      13      52%
==============================================================================

Configuration
ts   test_classes
api  reports/greetapi.sig
mode r
```

Real World Mode Without Fields and Enum Constants

The script executes the following command:

```
java -jar apicover.jar -ts test_classes -api reports/greetapi.sig
-apiinclude com.sun.tdk.samples.helloworld.api -mode r
-excludeFields -report reports/cov-realworld-without-fields.txt
```

In this case test coverage is 84%.

The report is named `reports/cov-realworld-without-fields.txt` and is shown below.

```
Coverage Report

================================================================================
Package                                  classes
    Class                                          members tested  %%
        Member
================================================================================
com                                      4           13      11      84%
--------------------------------------------------------------------------------
com.sun                                  4           13      11      84%
--------------------------------------------------------------------------------
com.sun.tdk                              4           13      11      84%
--------------------------------------------------------------------------------
com.sun.tdk.samples                      4           13      11      84%
--------------------------------------------------------------------------------
com.sun.tdk.samples.helloworld           4           13      11      84%
--------------------------------------------------------------------------------
com.sun.tdk.samples.helloworld.api       4           13      11      84%

    Greet                                            3       3       100%

    GreetFactory                                     1       1       100%

    Places                                           5       3       60%
      - getCountry()
      - valueOf(java.lang.String)

    TimeOfDay                                        4       4       100%
================================================================================
Overall coverage                         4           13      11      84%
================================================================================

Configuration
ts            test_classes
api           reports/greetapi.sig
excludeFields yes
mode          r
```

APPENDIX C

API Migration Compatibility Rules (Signature Test)

This appendix describes the rules used by the Signature Test tool API migration feature.

TABLE C-1describes the rules and shows whether they break only source compatibility or both source and binary compatibility. Comments and clarifications follow the table for rules that have an asterisk after their number

Note – A class is called *subclassable* if it has subclasses outside its package. In other words, this class is not final and has public constructor

TABLE C-1 API Migration Compatibility Rules

Rule #	Description	Breaks	Severity
	1 General Rules		
	The rules from this group are triggered if the more specific rules from the other groups below are not triggered. For example, adding an interface method defined in rule 2.1, adding constructor in the rule 5.5. In such cases more general rule 1.1 is ignored.		
1.1 *	API (public or protected) type (class, interface, enum, annotation type) or member (method, field) added	Source	Warning
1.2	API (public or protected) type (class, interface, enum, annotation type) or member (method, field) removed	Both	Severe
1.3	Narrowing type or member visibility - from public to non-public, from protected to package or private	Both	Severe
1.4 *	Generification of the public API type	None	
	2 Interfaces and Annotation Types		
2.1	Add methods	Both	Severe

TABLE C-1 API Migration Compatibility Rules

Rule #	Description	Breaks	Severity
2.2	Add fields	Both	Severe
2.3 *	Expand the set of superinterfaces (direct or indirect) if the added interface has a field (constant)	Source	Warning
2.4	Contract superinterface set (direct or inherited)	Both	Severe
2.5 *	Add member without default value to annotation type	Source	Severe
2.6 *	Add member with default value to annotation type	None	
2.7	Remove member from annotation type	Both	Severe
2.8	Remove default value from member of annotation type	Both	Severe
	3 Interfaces and Class Methods		
	Note: Some rules in this group apply only for class methods.		
3.1	Change signature and/or return type	Both	Severe
3.2	Change last parameter from array type T[] to variable array T...	None	
3.3	Change last parameter from array T... to array type T[]	Source	Severe
3.4	Change normalized throw list	Source	Severe
3.5	Decrease access from public to protected	Both	Severe
3.6 *	Increase access from protected to public if the class is subclassable	Source	Warning
3.7	Change method from abstract to non-abstract	None	
3.8	Change method from non-abstract to abstract (if the class can be subclassed)	Both	Severe
3.9	Change method from final to non-final	None	
3.10	Change method from non-final to final	Both	Severe
3.11	Change method from static to non-static	Both	Severe
3.12	Change method from non-static to static	Both	Severe
3.13	Change method from native to non-native	None	
3.14	Change method from non-native to native	None	
3.15	Change method from synchronized to non-synchronized	None	
3.16	Change method from non-synchronized to synchronized	None	
	4 Interfaces and Class Fields		
	Some rules from this group apply only for class fields.		
4.1	Change type	Both	Severe
4.2	Change/Remove constant value	Both	Warning

TABLE C-1 API Migration Compatibility Rules

Rule #	Description	Breaks	Severity
4.3	Decrease access	Both	Severe
4.4	Increase access	None	
4.5	Change from final to non-final	None	
4.6	Change from non-final to final	Both	Severe
4.7	Change from static to non-static	Bone	Severe
4.8	Change from non-static to static	Both	Severe
	5 Classes		
5.1 *	Add non-abstract and non-static methods	Both	Warning
5.2	Add abstract methods (if the class can be subclassed)	Both	Severe
5.3	Add static methods (if the class can be subclassed)	Both	Warning
5.4	Remove constructors	Both	Severe
5.5	Add first constructor with arguments or throws clause	Both	Severe
5.6	Add fields	Both	Severe
5.7	Expand implemented interface set (direct or indirect)		
5.7.1	The added interface adds abstract methods	Both	Severe
5.7.2	The new interface adds fields or inner classes	Source	Severe
5.7.3	If 5.7.1 and 5.7.3 are not true	None	
5.8	Contract implemented interface set (direct or indirect)	Both	Severe
5.9	Expand superclass set (direct or indirect)		
5.9.1	Add superclass adds abstract method (see rules 5.1 - 5.3)		
5.9.2	Add superclass adds field (see rule 5.6)		
5.9.3	Other cases		
5.10	Contract superclass set (direct or indirect)	Both	Severe
5.11	Change abstract to non-abstract	None	
5.12	Change non-abstract to abstract (if the class can be subclassed)	Both	Severe
5.13	Change final to non-final	None	
5.14	Change non-final to final	Both	Severe

Comments and Clarifications

Rule 1.1

Adding a class can theoretically break source code compatibility because new classes can be incorrectly resolved in an existing client's code with type-import-on-demand declarations (also know as wildcard imports). This can happen if the code uses another type with the same simple name. For example:

```
//client's code

import com.acme.*;
import com.client.*;

   .....

   Policy p = new Policy();   // this is com.client.Policy

   .....
```

In this case, after adding class `com.acme.Policy`, the compiler raises the error

```
reference to Policy is ambiguous, both class com.acme.Policy in
com.acme and class com.client.Policy in com.client match.
```

This rule is considered a warning because it will probably not affect binary compatibility and adhering to this rule makes API evolution very difficult.

Rule 1.4

Generics are a facility of generic programming that was added to the Java programming language as part of Java SE version 5.0. Generics allow a type or method to operate on objects of various types while providing compile-time type safety. Generification upgrades types using support to-be-specified-later types that are instantiated as needed for specific types that are provided as type parameters.

The Java programming language implements generics using erasure, which ensures that legacy and generic versions usually generate identical class files, except for some auxiliary information about types. Binary compatibility is not broken because it is possible to replace a legacy class file with a generic class file without changing or recompiling any client code.

To facilitate interfacing with non-generic legacy code, it is also possible to use the erasure of a parameterized type as a type. Such a type is called a raw type (*Java Language Specification* 3/4.8). Allowing the raw type also ensures backward compatibility for source code.

According to this, the following versions of the `java.util.Iterator` class are both binary and source code backward compatible:

- Class `java.util.Iterator` as it is defined in Java SE version 1.4:

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

- Class `java.util.Iterator` as it is defined in Java SE version 5.0:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Rule 2.3

Adding a superinterface with a constant field can shadow another entity with the same simple name. Consider the following code:

```
public interface Poet{
  boolean LITERATE = true;
}

public interface Playwright{
  boolean LITERATE = true;
}

public interface Shakespeare extends Poet {

}
```

Suppose that a new version of the Shakespeare interface implements interface Playwright as well as Poet as shown here:

```
public interface Shakespeare extends Poet, Playwright {

}
```

The following client code will not compile because the reference to `LITERATE` is ambiguous.

```
// client code
public class ShakespeareImpl implements Shakespeare {
    void introduce() {
        System.out.println("Hi, my name is Shakespeare and I'm " + LITERATE ?
            "quite literate" : "rather illiterate" );
    }
}
```

Rules 2.5, 2.6

The example below shows how adding a member without a default value to an annotation type breaks source code compatibility.

```
// annotation type v1
@interface Agent{
   String name();
}

// client's code which uses this annotation type v1
@Agent(name="James Bond")


// annotation type v2.1
@interface Agent{
   String name();
   String id(); // added member
}

// legacy code is not compilable due to
// error - annotation Agent is missing id
@Agent(name="James Bond")


// annotation type v2.2
@interface Agent{
   String name();
```

```
    String id() default "007";  // added member with default value
}

// legacy code is compilable
@Agent(name="James Bond")
```

Rule 3.6

Changing a method from protected to public can break source code compatibility if this method was overridden as protected. In this case the legacy code can not be recompiled because “access narrowing” is prohibited in the Java programming language. This rule is only a warning because it does not affect binary compatibility, and the probability of its affecting source code compatibility is very low.

Rule 4.2

Changing or removing constant values can break source code compatibility. For example, consider the following client code example. An integer constant named `NOTHING` with a value of 0 is used:

```
switch (i) {
    case NOTHING:
        // some actions
    case -1:
        // some other actions
}
```

Assume that the value of the constant `NOTHING` is changed from 0 to -1. The client code will not compile because of the duplicate case label. This rule is only a warning because it does not affect binary compatibility, and the probability of its affecting source code compatibility is very low.

Rule 5.1

Adding a regular method to a subclassable class can break source code and binary compatibility because a subclass can have a method with the same signature but with weaker access privileges. For example consider the following code example:

```
class ClientClass extends APIClass {
    private void foo() {}
}
```

Assume that the class `APIClass` is changed, and the method

```
protected void foo() {}
```

is added. The code cannot be recompiled due of an error that generates the following error message:

```
foo() in ClientClass cannot override foo() in APIClass; attempting to
assign weaker access privileges; was protected
```

Binary compatibility is broken as defined in JLS 3/3.14.12

Rule 5.3

Adding a static method to a subclassable class can break source code and binary compatibility, because a subclass can have a method with the same signature. For example consider the following code example:

```
class ClientClass extends APIClass {
    protected void foo() {}
}
```

Suppose that the class `APIClass` is changed and the method

```
protected static void foo() {}
```

is added. The code can not be recompiled due to an error that generates the following error message:

```
foo() in ClientClass cannot override foo() in APIClass; overridden
method is static
```

Binary compatibility is be broken as defined in JLS 3/3.14.12

Rule 5.7.2

As in the case of rule 2.3, adding a superinterface with a constant field can shadow another entity with the same simple name.

Index

U