TCK User’s Guide for Technology Implementors

Table of Contents

Eclipse Foundation

Technology Compatibility Kit User's Guide for Jakarta XML Binding

Release 3.0 for Jakarta EE

September 2020

References in this document to JAXB refer to the Jakarta XML Binding unless otherwise noted.

Preface

This guide describes how to install, configure, and run the Technology Compatibility Kit (TCK) that is used to test the Jakarta XML Binding (XML Binding 3.0) technology.

The XML Binding TCK is a portable, configurable automated test suite for verifying the compatibility of a vendor's implementation of the XML Binding 3.0 Specification (hereafter referred to as the vendor implementation or VI). The XML Binding TCK uses the JavaTest harness version 5.0 to run the test suite

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

Jakarta EE is a community sponsored and community run program. Organizations contribute, along side individual contributors who use, evolve and assist others. Commercial support is not available through the Eclipse Foundation resources. Please refer to the Eclipse EE4J project site (https://projects.eclipse.org/projects/ee4j). There, you will find additional details as well as a list of all the associated sub-projects (Implementations and APIs), that make up Jakarta EE and define these specifications. If you have questions about this Specification you may send inquiries to jaxb-dev@eclipse.org. If you have questions about this TCK, you may send inquiries to jaxb-dev@eclipse.org.

Who Should Use This Book

This guide is for vendors that implement the XML Binding 3.0 technology to assist them in running the test suite that verifies compatibility of their implementation of the XML Binding 3.0 Specification.

Before You Read This Book

You should be familiar with the XML Binding 3.0, version 3.0 Specification, which can be found at https://jakarta.ee/specifications/xml-binding/3.0.

Before running the tests in the XML Binding TCK, you should familiarize yourself with the JavaTest documentation which can be accessed at the JT Harness web site.

Typographic Conventions

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

Convention	Meaning	Example

Boldface	Boldface type indicates graphical user interface elements associated with an action, terms defined in text, or what you type, contrasted with onscreen computer output.	From the **File** menu, select **Open Project**.
		
		A **cache** is a copy that is stored locally.
		
		`machine_name% *su*`
		`Password:`

`Monospace`	Monospace type indicates the names of files and directories, commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
Italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.	Read Chapter 6 in the *User's Guide*. Do *not* save the file. The command to remove a file is `rm` *filename*.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`
Bash shell	`shell_name-shell_version$`
Bash shell for superuser	`shell_name-shell_version#`

1 Introduction

This chapter provides an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs) and describes the Jakarta XML Binding TCK (XML Binding 3.0 TCK). It also includes a high level listing of what is needed to get up and running with the XML Binding TCK.

This chapter includes the following topics:

1.1 Compatibility Testing

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

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

Compatibility test development for a given feature relies on a complete specification and compatible implementation (CI) for that feature. Compatibility testing is not primarily concerned with robustness, performance, nor ease of use.

1.1.1 Why Compatibility Testing is Important

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

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

all Jakarta platform ports.

1.1.2 TCK Compatibility Rules

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

1.1.3 TCK Overview

A TCK is a set of tools and tests used to verify that a vendor's compatible implementation of a Jakarta EE technology conforms to the applicable specification. All tests in the TCK are based on the written specifications for the Jakarta EE platform. A TCK tests compatibility of a vendor's compatible implementation of the technology to the applicable specification of the technology. Compatibility testing is a means of ensuring correctness, completeness, and consistency across all implementations developed by technology licensees.

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

The definition of required tests will change over time. Before your final certification test pass, be sure to download the latest version of this TCK.

1.1.4 Jakarta EE Specification Process (JESP) Program and Compatibility Testing

The Jakarta EE Specification Process (JESP) program is the formalization of the open process that has been used since 2019 to develop and revise Jakarta EE technology specifications in cooperation with the international Jakarta EE community. The JESP program specifies that the following three major components must be included as deliverables in a final Jakarta EE technology release under the direction of the responsible Expert Group:

- Technology Specification
- Compatible Implementation (CI)
- Technology Compatibility Kit (TCK)

For further information about the JESP program, go to Jakarta EE Specification Process community page https://jakarta.ee/specifications.

1.2 About the TCK

The XML Binding TCK 3.0 is designed as a portable, configurable, automated test suite for verifying the compatibility of a vendor's implementation of the XML Binding 3.0 Specification.

1.2.1 TCK Specifications and Requirements

This section lists the applicable requirements and specifications.

- Specification Requirements: Software requirements for a XML Binding implementation are described in detail in the XML Binding 3.0 Specification. Links to the XML Binding specification and other product information can be found at https://jakarta.ee/specifications/xml-binding/3.0.
- XML Binding Version: The XML Binding 3.0 TCK is based on the XML Binding Specification, Version 3.0.
- Compatible Implementation: One XML Binding 3.0 Compatible Implementation, Eclipse Implementation of JAXB 3.0.0 is available from the Eclipse EE4J project (https://projects.eclipse.org/projects/ee4j). See the CI documentation page at https://projects.eclipse.org/projects/ee4j.jaxb-impl for more information.

See the XML Binding TCK Release Notes for more specific information about Java SE version requirements, supported platforms, restrictions, and so on.

1.2.2 TCK Components

The XML Binding TCK 3.0 includes the following components:

- JavaTest harness version 5.0 and related documentation. See the JT Harness web site for additional information.
- XML Binding TCK signature tests; check that all public APIs are supported and/or defined as specified in the XML Binding Version 3.0 implementation under test.
- If applicable, an exclude list, which provides a list of tests that your implementation is not required to pass.
- API tests for all of the XML Binding API in all related packages:
  - `jakarta.xml.bind`
  - `jakarta.xml.bind.annotation`
  - `jakarta.xml.bind.annotation.adapters`

- `jakarta.xml.bind.attachment`
- `jakarta.xml.bind.helpers`
- `jakarta.xml.bind.util`

The XML Binding TCK tests run on the following platforms:

- CentOS Linux 7

1.2.3 JavaTest Harness

The JavaTest harness version 5.0 is a set of tools designed to run and manage test suites on different Java platforms. To JavaTest, Jakarta EE can be considered another platform. The JavaTest harness can be described as both a Java application and a set of compatibility testing tools. It can run tests on different kinds of Java platforms and it allows the results to be browsed online within the JavaTest GUI, or offline in the HTML reports that the JavaTest harness generates.

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

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

To run tests using the JavaTest harness, you specify which tests in the test suite to run, how to run them, and where to put the results as described in Chapter 4, "Setup and Configuration."

1.2.4 TCK Compatibility Test Suite

The test suite is the collection of tests used by the JavaTest harness to test a particular technology implementation. In this case, it is the collection of tests used by the XML Binding TCK 3.0 to test a XML Binding 3.0 implementation. The tests are designed to verify that a vendor's runtime implementation of the technology complies with the appropriate specification. The individual tests correspond to assertions of the specification.

The tests that make up the TCK compatibility test suite are precompiled and indexed within the TCK test directory structure. When a test run is started, the JavaTest harness scans through the set of tests that are located under the directories that have been selected. While scanning, the JavaTest harness

selects the appropriate tests according to any matches with the filters you are using and queues them up for execution.

1.2.5 Exclude Lists

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

A vendor's compatible implementation is not required to pass or run any test on the Exclude List. The Exclude List file, `${TS_HOME}/lib/jaxb_tck30.jtx`, is included in the XML Binding TCK.

From time to time, updates to the Exclude List are made available. The exclude list is included in the Jakarta TCK ZIP archive. Each time an update is approved and released, the version number will be incremented. You should always make sure you are using an up-to-date copy of the Exclude List before running the XML Binding TCK to verify your implementation.

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

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

In addition, all tests are run against the compatible implementations. Any tests that fail when run on a compatible Jakarta platform are put on the Exclude List. Any test that is not specification-based, or for which the specification is vague, may be excluded. Any test that is found to be implementation dependent (based on a particular thread scheduling model, based on a particular file system behavior, and so on) may be excluded.

Vendors are not permitted to alter or modify Exclude Lists. Changes to an Exclude List can only be made by using the procedure described in Section 2.3.1, "TCK Test Appeals Steps."

1.2.6 TCK Configuration

You need to set several variables in your test environment, modify properties in the `.jti` file you have

chosen, and then use the JavaTest harness to run the XML Binding tests, as described in Chapter 4, "Setup and Configuration."

The Jakarta EE Specification Process support multiple compatible implementations. These instructions explain how to get started with the Eclipse Implementation of JAXB 3.0.0 CI. If you are using another compatible implementation, refer to material provided by that implementation for specific instructions and procedures.

1.2.7 XML Binding Technology Overview

The XML Binding TCK 3.0 test suite comprises six test categories:

- Schema to Java mapping tests
- Java to schema mapping tests
- XML Binding framework API tests
- Content tree tests
- XML-to-Java name convention tests
- XML validation tests

Schema Compiler Tests

The schema compiler tests verify the following:

1. The schema compiler finds errors in invalid schema.
2. The schema compiler successfully compiles valid schema and the signatures of generated Java classes are appropriate.
3. The schema compiler appropriately compiles schema annotated with XML Binding custom binding declarations:
    - Successfully compiles valid source schema with well-formed binding declarations.
    - Fails to compile valid source schema with non-well-formed binding declarations.

XML Binding Framework API Tests

The XML Binding framework API tests cover public classes and interfaces defined in the `jakarta.xml.bind` package and sub-packages.

Content Tree Tests

The content tree tests test the classes generated by the XML Binding schema compiler, including:

1. Unmarshalling valid documents: the documents must be unmarshalled and validated correctly.

2. Marshalling documents: invokes marshalling for valid documents and checks that no errors are reported.
3. Content tree evaluating: check data in content tree after unmarshalling.
4. Content tree validating: modify data in content tree to make the tree invalid and, if the modification does not report an error, then enable XML Binding 3.0 validation on Marshaller, and verify that Marshaller reports the error.

XML-to-Java Name Conversion Tests

The XML-to-Java name conversion tests check that XML-to-Java name conversion rules are implemented correctly and support most of the Unicode characters that are allowed for XML names.

Schema Generator Tests

The schema generator tests verify that:

1. The schema generator supports all the Java Types to XML mapping.
2. The schema generator appropriately maps Java program elements to XML Schema constructs.
3. The schema generator appropriately maps Java program elements that are customized using mapping annotations and program annotations that are based on JSR–175 program annotation facility:
   - Successfully generates valid target schema from correctly annotated Java program element.
   - Fails to process annotated Java program elements that have constraint violations as specified in the `Java Types to XML` section of the XML Binding 3.0 specification and `jakarta.xml.bind.annotation` javadocs.
   - The schema generator successfully produces a well-formed schema; all documents expected to be valid against generated schema are valid and all documents expected to be invalid against generated schema are invalid.

XML Validation Tests

XML Validation tests are designed to check schema validation when a mechanisms other than JAXP is used. Note that XML validation tests are mandatory when XML validation is not performed via JAXP.

1.3 Getting Started With the TCK

This section provides an general overview of what needs to be done to install, set up, test, and use the XML Binding TCK. These steps are explained in more detail in subsequent chapters of this guide.

1. Make sure that the following software has been correctly installed on the system hosting the JavaTest harness:

- Java SE 8
- A CI for XML Binding 3.0. One example is Eclipse Implementation of JAXB 3.0.0.
- XML Binding TCK version 3.0, which includes:
  - The Checker Framework: checker-3.5.0.jar
- The XML Binding 3.0 Vendor Implementation (VI)
  See the documentation for each of these software applications for installation instructions. See Chapter 3, "Installation," for instructions on installing the XML Binding TCK.

  1. Set up the XML Binding TCK software.
     See Chapter 4, "Setup and Configuration," for details about the following steps.
     1. Set up your shell environment.
     2. Choose proper `.jti` configuration file in the `${TS_HOME}/lib` directory.
     3. Modify the required properties in the `.jti` file you have chosen.
  2. Test the XML Binding 3.0 implementation.
     Test the XML Binding implementation installation by running the test suite. See Chapter 5, "Executing Tests."

2 Procedure for Certification

This chapter describes the compatibility testing procedure and compatibility requirements for Jakarta XML Binding. This chapter contains the following sections:

2.1 Certification Overview

The certification process for XML Binding 3.0 consists of the following activities:

- Install the appropriate version of the Technology Compatibility Kit (TCK) and execute it in accordance with the instructions in this User's Guide.
- Ensure that you meet the requirements outlined in Compatibility Requirements below.
- Certify to the Eclipse Foundation that you have finished testing and that you meet all of the compatibility requirements, as required by the Eclipse Foundation TCK License.

2.2 Compatibility Requirements

The compatibility requirements for XML Binding 3.0 consist of meeting the requirements set forth by the rules and associated definitions contained in this section.

2.2.1 Definitions

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

Table 2-1 Definitions

Term	Definition
API Definition Product	A Product for which the only Java class files contained in the product are those corresponding to the application programming interfaces defined by the Specifications, and which is intended only as a means for formally specifying the application programming interfaces defined by the Specifications.
Computational Resource	A piece of hardware or software that may vary in quantity, existence, or version, which may be required to exist in a minimum quantity and/or at a specific or minimum revision level so as to satisfy the requirements of the Test Suite. Examples of computational resources that may vary in quantity are RAM and file descriptors. Examples of computational resources that may vary in existence (that is, may or may not exist) are graphics cards and device drivers. Examples of computational resources that may vary in version are operating systems and device drivers.
Configuration Descriptor	Any file whose format is well defined by a specification and which contains configuration information for a set of Java classes, archive, or other feature defined in the specification.
Conformance Tests	All tests in the Test Suite for an indicated Technology Under Test, as released and distributed by the Eclipse Foundation, excluding those tests on the published Exclude List for the Technology Under Test.
Container	An implementation of the associated Libraries, as specified in the Specifications, and a version of a Java Platform, Standard Edition Runtime Product, as specified in the Specifications, or a later version of a Java Platform, Standard Edition Runtime Product that also meets these compatibility requirements.
Documented	Made technically accessible and made known to users, typically by means such as marketing materials, product documentation, usage messages, or developer support programs.
Exclude List	The most current list of tests, released and distributed by the Eclipse Foundation, that are not required to be passed to certify conformance. The Jakarta EE Specification Committee may add to the Exclude List for that Test Suite as needed at any time, in which case the updated TCK version supplants any previous Exclude Lists for that Test Suite.

Term	Definition
Libraries	The class libraries, as specified through the Jakarta EE Specification Process (JESP), for the Technology Under Test. The Libraries for Jakarta XML Binding are listed at the end of this chapter.
Location Resource	A location of classes or native libraries that are components of the test tools or tests, such that these classes or libraries may be required to exist in a certain location in order to satisfy the requirements of the test suite. For example, classes may be required to exist in directories named in a CLASSPATH variable, or native libraries may be required to exist in directories named in a PATH variable.
Maintenance Lead	The corresponding Jakarta EE Specification Project is responsible for maintaining the Specification, and the TCK for the Technology. The Specification Project Team will propose revisions and updates to the Jakarta EE Specification Committee which will approve and release new versions of the specification and TCK.
Operating Mode	Any Documented option of a Product that can be changed by a user in order to modify the behavior of the Product. For example, an Operating Mode can be binary (enable/disable optimization), an enumeration (select from a list of protocols), or a range (set the maximum number of active threads). Note that an Operating Mode may be selected by a command line switch, an environment variable, a GUI user interface element, a configuration or control file, etc.
Product	A vendor’s product in which the Technology Under Test is implemented or incorporated, and that is subject to compatibility testing.
Product Configuration	A specific setting or instantiation of an Operating Mode. For example, a Product supporting an Operating Mode that permits user selection of an external encryption package may have a Product Configuration that links the Product to that encryption package.
Rebuildable Tests	Tests that must be built using an implementation-specific mechanism. This mechanism must produce specification-defined artifacts. Rebuilding and running these tests against a known compatible implementation verifies that the mechanism generates compatible artifacts.
Resource	A Computational Resource, a Location Resource, or a Security Resource.

Term	Definition
Rules	These definitions and rules in this Compatibility Requirements section of this User's Guide.
Runtime	The Containers specified in the Specifications.
Security Resource	A security privilege or policy necessary for the proper execution of the Test Suite. For example, the user executing the Test Suite will need the privilege to access the files and network resources necessary for use of the Product.
Specifications	The documents produced through the Jakarta EE Specification Process (JESP) that define a particular Version of a Technology. The Specifications for the Technology Under Test are referenced later in this chapter.
Technology	Specifications and one or more compatible implementations produced through the Jakarta EE Specification Process (JESP).
Technology Under Test	Specifications and a compatible implementation for Jakarta XML Binding Version 3.0.
Test Suite	The requirements, tests, and testing tools distributed by the Maintenance Lead as applicable to a given Version of the Technology.
Version	A release of the Technology, as produced through the Jakarta EE Specification Process (JESP).
Schema Compiler	A schema compiler generates Java source files or compiled classes that represent XML data described by W3C XML Schema recommendation [XSD Part 1][XSD Part 2].
Schema Compiler Output	The output of a Schema Compiler intended to be run in combination with the Libraries.
Schema Generator	A software development tool that maps a set of existing Java sources or class files to Schema Generator Output. The mapping is described by program annotations. The Schema Generator can be invoked directly by a programmer without programming.
Schema Generator Output	The output of a Schema Generator intended to be a W3C XML Schema defined in the XML Schema recommendation [XSD Part 1][XSD Part 2]. The output of a Schema Generator that is in the format defined by the W3C XML Schema recommendation [XSD Part 1][XSD Part 2].

2.2.2 Rules for Jakarta XML Binding Products

The following rules apply for each version of an operating system, software component, and hardware platform Documented as supporting the Product:

XmlBinding1 The Product must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests, in every Product Configuration and in every combination of Product Configurations, except only as specifically exempted by these Rules.

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

XmlBinding1.1 If an Operating Mode controls a Resource necessary for the basic execution of the Test Suite, testing may always use a Product Configuration of that Operating Mode providing that Resource, even if other Product Configurations do not provide that Resource. Notwithstanding such exceptions, each Product must have at least one set of Product Configurations of such Operating Modes that is able to pass all the Conformance Tests.

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

XmlBinding1.2 A Product Configuration of an Operating Mode that causes the Product to report only version, usage, or diagnostic information is exempted from these compatibility rules.

XmlBinding1.3 An API Definition Product is exempt from all functional testing requirements defined here, except the signature tests.

XmlBinding2 Some Conformance Tests may have properties that may be changed. Properties that can be changed are identified in the configuration interview. Properties that can be changed are identified in the JavaTest Environment (.jte) files in the Test Suite installation. Apart from changing such properties and other allowed modifications described in this User's Guide (if any), no source or binary code for a Conformance Test may be altered in any way without prior written permission. Any such allowed alterations to the Conformance Tests will be provided via the Jakarta EE Specification Project website and apply to all vendor compatible implementations.

XmlBinding3 The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

XmlBinding4 The Exclude List associated with the Test Suite cannot be modified.

XmlBinding5 The Maintenance Lead can define exceptions to these Rules. Such exceptions would be made available as above, and will apply to all vendor implementations.

XmlBinding6 All hardware and software component additions, deletions, and modifications to a Documented supporting hardware/software platform, that are not part of the Product but required for

the Product to satisfy the compatibility requirements, must be Documented and available to users of the Product.

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

XmlBinding7 The Product must contain the full set of public and protected classes and interfaces for all the Libraries. Those classes and interfaces must contain exactly the set of public and protected methods, constructors, and fields defined by the Specifications for those Libraries. No subsetting, supersetting, or modifications of the public and protected API of the Libraries are allowed except only as specifically exempted by these Rules.

XmlBinding7.1 If a Product includes Technologies in addition to the Technology Under Test, then it must contain the full set of combined public and protected classes and interfaces. The API of the Product must contain the union of the included Technologies. No further modifications to the APIs of the included Technologies are allowed.

XmlBinding8 Except for tests specifically required by this TCK to be rebuilt (if any), the binary Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

XmlBinding9 The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.

Rules for Products that include Support for Software Development

The following Rules, in addition to the Rules for XML Binding 3.0 Products, apply for each version of an operating system, software component, and hardware platform Documented as supporting the Product:

XmlBindingSD1 For each Product Configuration that causes the Schema Compiler to produce Schema Compiler Output, the Product must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests, except only as specifically exempted by these Rules.

XmlBindingSD2 For each Product Configuration that causes the Schema Compiler to produce Schema Compiler Output, in cases where such Schema Compiler Output contains only value classes, that Schema Compiler Output, when combined with the XML Binding Compatibile Implementation Libraries, must execute properly when run on all Java SE Reference Runtime versions that correspond to versions of Java SE Documented as supporting the Product.

XmlBindingSD3 The Schema Compiler must not produce Schema Compiler Output from schema that do not conform to the W3C XML Schema recommendation [XSD Part 1][XSD Part 2].

XmlBindingSD3.1 The Schema Compiler in non-default Product Configurations may produce Schema Compiler Output when an XML Schema contains constructs for which a binding has not been defined by the Specification.

XmlBindingSD3.2 The Schema Compiler in non-default Product Configurations may produce Schema Compiler Output when a custom binding declaration is encountered.

XmlBindingSD4 The Schema Compiler Output of the Schema Compiler must be either in class format defined by the Java Virtual Machine (JVM) Specifications or in source file format defined by Java Language Specification (JLS).

XmlBindingSD5 For each Product Configuration that causes the Schema Generator to produce Schema Generator Output, the Product must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests, except only as specifically exempted by these Rules.

XmlBindingSD6 For each Product Configuration that causes the Schema Generator to produce Schema Generator Output, the Output of Schema Generator must fully meet W3C requirements for the XML schema language.

XmlBindingSD7 The Schema Generator must not produce Schema Generator Output when a program element violates mapping constraints defined by Specifications.

XmlBindingSD8 The Schema Generator may produce Schema Generator Output when a program element contains constructs for which a mapping has not been defined by the Specification.

XmlBindingSD9 A schema produced by the Schema Generator from source code containing a comment or a directive, or from a class file containing non-standard attributes, must be equivalent to a schema produced by the Schema Generator from the same source code with the comment or directive removed, or from the class file with the attributes removed. Two schemas are equivalent if and only if they validate the same set of XML documents (any XML document is either valid against both schemas or invalid against both schemas).

XmlBindingSD10 Except for tests specifically required by this TCK to be modified (if any) the source for Schema Compiler and Schema Generator Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.

2.3 Test Appeals Process

Jakarta has a well established process for managing challenges to its TCKs. Any implementor may submit a challenge to one or more tests in the XML Binding TCK as it relates to their implementation. Implementor means the entity as a whole in charge of producing the final certified release. **Challenges filed should represent the consensus of that entity**.

2.3.1 Valid Challenges

Any test case (e.g., test class, @Test method), test case configuration (e.g., deployment descriptor), test beans, annotations, and other resources considered part of the TCK may be challenged.

The following scenarios are considered in scope for test challenges:

- Claims that a test assertion conflicts with the specification.
- Claims that a test asserts requirements over and above that of the specification.
- Claims that an assertion of the specification is not sufficiently implementable.
- Claims that a test is not portable or depends on a particular implementation.

2.3.2 Invalid Challenges

The following scenarios are considered out of scope for test challenges and will be immediately closed if filed:

- Challenging an implementation's claim of passing a test. Certification is an honor system and these issues must be raised directly with the implementation.
- Challenging the usefulness of a specification requirement. The challenge process cannot be used to bypass the specification process and raise in question the need or relevance of a specification requirement.
- Claims the TCK is inadequate or missing assertions required by the specification. See the Improvement section, which is outside the scope of test challenges.
- Challenges that do not represent a consensus of the implementing community will be closed until such time that the community does agree or agreement cannot be made. The test challenge process is not the place for implementations to initiate their own internal discussions.
- Challenges to tests that are already excluded for any reason.
- Challenges that an excluded test should not have been excluded and should be re-added should be opened as a new enhancement request

Test challenges must be made in writing via the XML Binding specification project issue tracker as described in Section 2.3.3, "TCK Test Appeals Steps."

All tests found to be invalid will be placed on the Exclude List for that version of the XML Binding TCK.

2.3.3 TCK Test Appeals Steps

1. Challenges should be filed via the Jakarta XML Binding specification project's issue tracker using the label `challenge` and include the following information:
    - The relevant specification version and section number(s)
    - The coordinates of the challenged test(s)
    - The exact TCK and exclude list versions
    - The implementation being tested, including name and company

- The full test name
- A full description of why the test is invalid and what the correct behavior is believed to be
- Any supporting material; debug logs, test output, test logs, run scripts, etc.

2. Specification project evaluates the challenge.
   Challenges can be resolved by a specification project lead, or a project challenge triage team, after a consensus of the specification project committers is reached or attempts to gain consensus fails. Specification projects may exercise lazy consensus, voting or any practice that follows the principles of Eclipse Foundation Development Process. The expected timeframe for a response is two weeks or less. If consensus cannot be reached by the specification project for a prolonged period of time, the default recommendation is to exclude the tests and address the dispute in a future revision of the specification.
3. Accepted Challenges.
   A consensus that a test produces invalid results will result in the exclusion of that test from certification requirements, and an immediate update and release of an official distribution of the TCK including the new exclude list. The associated `challenge` issue must be closed with an `accepted` label to indicate it has been resolved.
4. Rejected Challenges and Remedy.
   When a `challenge` issue is rejected, it must be closed with a label of `invalid` to indicate it has been rejected. There appeal process for challenges rejected on technical terms is outlined in Escalation Appeal. If, however, an implementer feels the TCK challenge process was not followed, an appeal issue should be filed with specification project's TCK issue tracker using the label `challenge-appeal`. A project lead should escalate the issue with the Jakarta EE Specification Committee via email (jakarta.ee-spec@eclipse.org). The committee will evaluate the matter purely in terms of due process. If the appeal is accepted, the original TCK challenge issue will be reopened and a label of `appealed-challenge` added, along with a discussion of the appeal decision, and the `challenge-appeal` issue with be closed. If the appeal is rejected, the `challenge-appeal` issue should closed with a label of `invalid`.
5. Escalation Appeal.
   If there is a concern that a TCK process issue has not been resolved satisfactorily, the Eclipse Development Process Grievance Handling procedure should be followed to escalate the resolution. Note that this is not a mechanism to attempt to handle implementation specific issues.

2.4 Specifications for Jakarta XML Binding

The Jakarta XML Binding specification is available from the specification project web-site: https://jakarta.ee/specifications/xml-binding/3.0.

2.5 Libraries for Jakarta XML Binding

The following is a list of the packages comprising the required class libraries for XML Binding 3.0:

- `jakarta.xml.bind`
- `jakarta.xml.bind.annotation`
- `jakarta.xml.bind.annotation.adapters`
- `jakarta.xml.bind.attachment`
- `jakarta.xml.bind.helpers`
- `jakarta.xml.bind.util`

For the latest list of packages, also see:

https://jakarta.ee/specifications/xml-binding/3.0

3 Installation

This chapter explains how to install the Jakarta XML Binding TCK software.

After installing the software according to the instructions in this chapter, proceed to Chapter 4, "Setup and Configuration," for instructions on configuring your test environment.

3.1 Obtaining a Compatible Implementation

Each compatible implementation (CI) will provide instructions for obtaining their implementation. Eclipse Implementation of JAXB 3.0.0 is a compatible implementation which may be obtained from https://projects.eclipse.org/projects/ee4j.jaxb-impl

3.2 Installing the Software

Before you can run the XML Binding TCK tests, you must install and set up the following software components:

- Java SE 8
- A CI for XML Binding 3.0, one example is Eclipse Implementation of JAXB 3.0.0
- XML Binding TCK version 3.0, which includes:
  - The Checker Framework: checker-3.5.0.jar
- The XML Binding 3.0 Vendor Implementation (VI)

Follow these steps:

1. Install the Java SE 8 software, if it is not already installed.
   Download and install the Java SE 8 software from http://www.oracle.com/technetwork/java/javase/downloads/index.html. Refer to the installation instructions that accompany the software for additional information.
2. Install the XML Binding TCK 3.0 software.
   1. Copy or download the XML Binding TCK software to your local system.
      You can obtain the XML Binding TCK software from the Jakarta EE site https://jakarta.ee/specifications/xml-binding/3.0.
   2. Use the `unzip` command to extract the bundle in the directory of your choice:
      `unzip jakarta-xml-binding-tck-3.0.0.zip`
      This creates the TCK directory. The TCK is the test suite home, `<TS_HOME>`.
3. Install a XML Binding 3.0 Compatible Implementation.

A Compatible Implementation is used to validate your initial configuration and setup of the XML Binding TCK 3.0 tests, which are explained further in Chapter 4, "Setup and Configuration."
The Compatible Implementations for XML Binding are listed on the Jakarta EE Specifications web site: https://jakarta.ee/specifications/xml-binding/3.0.

4. Install the XML Binding VI to be tested.
   Follow the installation instructions for the particular VI under test.

4 Setup and Configuration

The Jakarta EE Specification process provides for any number of compatible implementations. As additional implementations become available, refer to project or product documentation from those vendors for specific TCK setup and operational guidance.

This chapter describes how to set up the XML Binding TCK and JavaTest harness software. Before proceeding with the instructions in this chapter, be sure to install all required software, as described in Chapter 3, "Installation."

After completing the instructions in this chapter, proceed to Chapter 5, "Executing Tests," for instructions on running the XML Binding TCK.

4.1 Configuring the XML Binding TCK

4.1.1 TCK Configuration Concepts

XML Binding 3.0 comprises of a schema compiler and a schema generator. Both are types of compilers; the former maps from a schema to Java, and the latter maps from Java to schema. For the purposes of brevity in these instructions, the term compiler refers to both the schema compiler and the schema generator.

Configuring Your Environment to Run the TCK Against the Compatible Implementation

1. Set the following environment variables in your shell environment:
    1. `JAVA_HOME` to the directory in which Java SE 8 is installed
    2. `TS_HOME` to the directory in which the XML Binding TCK software is installed
    3. `JAXB_HOME` to the directory in which the XML Binding software is installed
    4. `PATH` to include the following directories: `$JAVA_HOME/bin` and `$ANT_HOME/bin`
2. Edit `$TS_HOME/testsuite.jtt` and modify `finder` settings to contain proper XML Binding TCK software installation directory path prefix
3. Choose XML Binding TCK software configuration file template to use:
    1. `$TS_HOME/lib/javasoft-multiJVM.jti` to run tests in multi JVM mode (this is the default configuration template)
    2. or `$TS_HOME/lib/javasoft-singleJVM.jti` to run tests in single JVM mode

4. Edit chosen file, for example `${TS_HOME}/lib/javasoft-multiJVM.jti`, and set environment variables and configuration properties.
    1. Set `TESTSUITE` environment variable to contain XML Binding TCK software installation directory.
    2. Set `WORKDIR` environment variable to contain XML Binding TCK software working directory. This directory will contain raw testing data and results.
    3. Set `jck.env.jaxb.classes.jaxbClasses` property to contain The Checker Framework library and all XML Binding libraries:
        - checker.jar
        - `JAXB_HOME`/mod/jakarta.activation.jar
        - `JAXB_HOME`/mod/jakarta.xml.bind-api.jar
        - `JAXB_HOME`/mod/jaxb-core.jar
        - `JAXB_HOME`/mod/jaxb-impl.jar
        - `JAXB_HOME`/mod/jaxb-jxc.jar
    4. Set `jck.env.jaxb.testExecute.cmdAsFile` property to contain Java SE 8 installation Directory.
    5. Set `jck.env.jaxb.testExecute.otherEnvVars` property to contain
        - `JAVA_HOME` variable with Java SE 8 installation directory,
        - `JAXB_HOME` variable with XML Binding installation directory,
    6. Set `jck.env.jaxb.schemagen.run.jxcCmd` property to contain proper schema generator script for your platform.
    7. Set `jck.env.jaxb.xsd_compiler.testCompile.xjcCmd` property to contain proper schema compiler script for your platform. This property can also affect which CI is used in tests. For example EclipseLink MOXy also generates with model classes `jaxb.properties` file with following content `jakarta.xml.bind.JAXBContextFactory=org.eclipse.persistence.jaxb.JAXBContextFactory`.
    8. Set `jck.env.jaxb.xsd_compiler.skipValidationOptional` property to contain value `Yes`. XML validation is implemented from JAXP in XML Binding 3.0 Compatible Implementation. See Section, "Configuration Properties" for details.

Note: More info about JAXBContext loading in Discovery of JAXB implementation.

An example of modified `${TS_HOME}/lib/javasoft-multiJVM.jti` file:

```
TESTSUITE=/workspace/XMLB-TCK-3.0
WORKDIR=/workspace/XMLB-TCK-3.0/batch-multiJVM/work
jck.concurrency.concurrency=1
jck.env.description=multi jvm
jck.env.envName=multi_jvm
jck.env.jaxb.classes.jaxbClasses=/data/tck/tmp/checker.jar
  /data/tck/tmp/jaxb-ri/mod/jakarta.activation.jar
  /data/tck/tmp/jaxb-ri/mod/jakarta.xml.bind-api.jar
  /data/tck/tmp/jaxb-ri/mod/jaxb-core.jar
  /data/tck/tmp/jaxb-ri/mod/jaxb-impl.jar
  /data/tck/tmp/jaxb-ri/mod/jaxb-jxc.jar
jck.env.jaxb.schemagen.run.jxcCmd=/bin/sh
  /workspace/XMLB-TCK-3.0/linux/bin/schemagen.sh
jck.env.jaxb.schemagen.skipJ2XOptional=Yes
jck.env.jaxb.testExecute.cmdAsFile=/opt/jdk1.8.0_192.jdk/bin/java
jck.env.jaxb.testExecute.otherEnvVars=JAVA_HOME\=/opt/jdk1.8.0_192.jdk
  JAXB_HOME\=/workspace/jaxb-ri
jck.env.jaxb.testExecute.otherOpts=
jck.env.jaxb.xsd_compiler.defaultOperationMode=Yes
jck.env.jaxb.xsd_compiler.skipValidationOptional=Yes
jck.env.jaxb.xsd_compiler.testCompile.xjcCmd=/bin/sh
  /workspace/XMLB-TCK-3.0/linux/bin/xjc.sh
jck.env.testPlatform.local=Yes
jck.env.testPlatform.multiJVM=Yes
jck.excludeList.latestAutoCheck=No
jck.excludeList.latestAutoCheckInterval=7
jck.excludeList.latestAutoCheckMode=everyXDays
jck.excludeList.needExcludeList=No
jck.keywords.keywords.mode=expr
jck.keywords.needKeywords=No
jck.priorStatus.needStatus=No
jck.priorStatus.status=
jck.tests.needTests=No
jck.tests.tests=
jck.tests.treeOrFile=tree
jck.timeout.timeout=1
```

4.1.2 XML Binding-Specific Configuration Settings

A number of properties in the JavaTest harness configuration file are generic and apply to most Java product TCK suites. This section describes configuration settings that are specific to the XML Binding TCK.

The XML Binding TCK 3.0 includes a special configuration properties used by the JavaTest harness to collect information from you about your test environment and to create the configuration used to run tests. The TCK configuration properties consists of the following types:

- **Test Environment**: the first type of the properties collects information about the test environment used to run the XML Binding TCK 3.0 (such as the type of environment and the type of agent used).
- **Schema Compiler Questions**: the second type of the properties collects information about how to compile the source schema on your platform.
  - XML Validation Tests: optional tests that must be run if schema validation is performed using a mechanism other than JAXP.
- **Schema Generator Questions**: the type section of the properties collects information about how to generate XML schema from Java sources on your platform.
- **How Tests are Run**: the last section of the properties collects information on how the tests are run (such as Keywords and the Exclude List).

4.1.3 Configuration Properties

Following list describes the configuration properties and gives examples of relevant values to correctly configure the JavaTest harness to run the test suite.

Configuration Name

Property: jck.env.envName

Provide a short identifier that names the configuration you are creating.

This is a short string that identifies this particular configuration. Select a name that describes the configuration.

Example: `jaxb_win32`

The name must begin with a letter, followed by letters, digits, or an underscore, and must not contain a white space or punctuation characters.

Description

Property: jck.env.description

Provide a short description that identifies the configuration you are creating.

JavaTest uses this short description to provide more detail about the configuration, for example, in reports. This information might be useful to an auditor looking at the test run reports, and could include the version number of the product and the name of the tester.

Example: `John Smith x86 Eclipse Implementation of JAXB 3.0.0`

Execution Mode

Property: jck.env.testPlatform.multiJVM

Specify whether you want to run the tests in SingleJVM or MultiJVM mode. Depending on the selected mode, you can configure to run the compiler either from the command line, or via the Java Compiler API, or via a custom class accessed by the core reflection API.

Answer `MultiJVM Mode` if the JavaTest harness can directly start your product, for example, from a command line in an MSDOS window or UNIX shell; this is the required mode for the XML Binding TCK.

Typically, applications that do not support such actions run in point-and-click environments or in browsers. For example, the JavaTest harness cannot start a new instance of a Java virtual machine inside a browser.

If you answer MultiJVM Mode , then you need to answer additional interview questions to specify whether you plan to run the tests locally. In addition, you need to specify the details about the Java Virtual Machine in which the tests will run (the questions in the **Runtime Other JVM** section of this interview) and the details about the Java launcher.

Execute Locally

Property: jck.env.testPlatform.local

Will you run the TCK tests on the system that hosts the JavaTest harness?

If you plan to run the tests on the system on which you are running this interview, then the editor gathers configuration information directly from the system. For example, you will be able to browse the local file system to point to the required files.

Agent Type

Property: jck.env.jaxb.agent.agentType

The JavaTest Agent can be run in two modes: active and passive. Which mode do you wish to use?

Select the agent type you wish to use. If you want to use the passive agent, then additional properties must be provided to specify the passive agent host and default agent port.

Passive Host

Property: jck.env.jaxb.agent.agentPassiveHost

What is the of the host on which you run the tests?

This is the name of the system on which the passive agent runs and executes tests. The name of the system must be accessible from the system on which JavaTest runs. Since the JavaTest agent is passive,

you can only change the machine you specify here by running this interview again and changing the name of the passive host.

Default Agent Port

Property: jck.env.jaxb.agent.useAgentPortDefault

Do you wish to use the default port for the JavaTest harness to contact the JavaTest Agent?

By default, the JavaTest harness communicates with a passive agent using port 1908. You must change only the default if you are running more than one passive agent on a system; this might be the case if you have set up batch runs.

If you answer `No`, then you will need to specify the default port number in the next interview question.

Passive Port

Property: jck.env.jaxb.agent.agentPassivePort

Which port does the JavaTest harness use to contact the JavaTest Agent?

Specify the port you want to use to communicate with the JavaTest agent.

Test Platform File Separator

Property: jck.env.jaxb.agent.fileSeparator

Which file separator character is used on the test platform?

Specify the appropriate file separator for your operating system.

For example, for Windows, use the back slash `\` character. For Solaris OS or Linux operation system, use the forward slash `/` character.

Test Platform Path Separator

Property: jck.env.jaxb.agent.pathSeparator

Which path separator character is used on the test platform?.

Specify the appropriate path separator for your operating system.

For example, on Windows systems, specify the `;` character, on Unix systems specify the `:` character. Other platforms may differ.

Agent Map File

Property: jck.env.jaxb.agent.mapArgs

Because JavaTest and the JavaTest agent run on different systems, values such as path names and variable names may differ on each system. For example, if a file is accessed over a network, it might be mounted as `H:\tests\lib` on the JavaTest system and as `F:\tests\lib` on the system running the JavaTest agent. The map file is used to associate values used on the JavaTest agent with those used by JavaTest. The map file is specified on the JavaTest agent, but the use of the map must be enabled here. For more information about map files, refer to the JavaTest User's Guide or to the JavaTest online help.

Java Launcher

Property: jck.env.jaxb.testExecute.cmdAsFile

Enter the full path name of the Java launcher that runs the tests through the Java Compiler API interface.

This property is required only if `jck.env.testPlatform.multiJVM` property value has been set to `Yes`.

This is the command used on the reference system to invoke the `java` command.

Use the `java` command in JDK (not JRE).

- On Solaris or Linux platforms, use expanded value of: `$JAVA_HOME/bin/java`
  - For example: `/usr/java/bin/java`
- On Win32 systems, use expanded value of: `%JAVA_HOME%\bin\java.exe`
  - For example: `C:\Java\jdk-9\bin\java.exe`

Ensure that you supply the full file name, including the file extension.

Other Options

Property: jck.env.jaxb.testExecute.otherOpts

Specify any additional commandline options that must be set.

Specify any command-line options that JavaTest should set for your compiler when it runs the tests on your system. Specify the options using `name=value` pairs. For example you may want to specify: `-Xmx256m -Xms128m` to change the default maximum and initial Java heap size.

Other Environment Variables

Property: jck.env.jaxb.testExecute.otherEnvVars

Except for CLASSPATH, if there are any other environment variables that must be set for the reference Java launcher, enter them here.

Specify any environment variables that the JavaTest harness should use in the environment created to run the tests on your system. Specify the variables using the `name=value` pairs.

You must specify at least `JAVA_HOME` and `JAXB_HOME`.

The JavaTest cannot inherit environment variables from your operating system environment. If you find that your tests run correctly outside of the JavaTest harness, but do not run correctly using the JavaTest harness, then the reason might be that a value set in your operating system environment is not available to the test. If you check your operating system environment and determine that this is the case, then set the missing values here.

Need Extra XML Binding Classes

Property: jck.env.jaxb.classes.needJaxbClasses

Do you need to specify extra XML Binding classes for the Java launcher to execute tests?

The XML Binding classes must be available (that is, can be found and loaded) during test execution. This means that XML Binding JAR archives which contain the XML Binding classes must be on the class path of the the Java VM executing the test.

Answer `No` if the XML Binding JAR archives are in the Java default class path, for example, all jar files are stored in the jdk_install_dir/jre/lib/ext directory.

Answer `Yes` if some directories or archives are not located on the Java VM default class path.

If you answer `Yes`, then must specify necessary XML Binding archives.

Extra XML Binding Classes

Property: jck.env.jaxb.classes.jaxbClasses

Extra XML Binding classes to be put on class path of the the Java VM executing the test.

Specify all necessary XML Binding archives required for tests execution. Archive files must be specified with full path and separated by space.

Schema Compiler Operation Mode

Property: jck.env.jaxb.xsd_compiler.defaultOperationMode

Would you like to run your schema compiler in default operation mode?

A XML Binding compatible implementation must support a default operating mode in which all the required XML Schema to Java bindings that are described in the XML Binding specification must be implemented. Errors must be reported when alternative or extension non-required features are encountered.

A XML Binding compatible implementation may support non-default operating modes. These modes must support all the XML Schema to Java bindings that are described in XML Binding specification, but may also generate alternative or extension bindings for XML Schema constructs, which are not required by this specification.

If you select `Yes`, then the schema compiler is supposed to run in default operation mode and the tests for non-required features must be passed (as negative tests with expected error messages).

If you select `No`, then the schema compiler is supposed to run in non-default operation mode and the tests for non-required features will not be run. For more information on default operation mode requirements, see the Compatibility section of the XML Binding specification.

Optional XML Validation Tests

Property: jck.env.jaxb.xsd_compiler.skipValidationOptional

Is Schema Validation implemented via mechanism different from JAXP?

XML validation can be implemented from JAXP or from some other mechanism. If the validation is implemented from JAXP, then validation checking tests can be skipped by deselecting them in the test interview. If you use some other tools for schema validation, then XML validation tests are mandatory.

Schema Compiler Run Command

Property: jck.env.jaxb.xsd_compiler.testCompile. xjcCmd

Enter the command to run the a schema compiler.

This is the command used to invoke your schema compiler. For example, for the Eclipse XML Binding implementation on Linux we provide a script `$TS_HOME/linux/bin/xjc.sh`, which compiles schemas into class files. The command is as follows:

```
/bin/sh /workspace/XMLB-TCK-3.0/linux/bin/xjc.sh
```

For the Eclipse XML Binding implementation on Windows, we provide a script `%TS_HOME%\win32\bin\xjc.bat`, which compiles schemas into class files. The command is as follows:

```
c:\workspace\XMLB-TCK-3.0\win32\bin\xjc.bat
```

The xjc script supports the following mandatory command-line options: `xjc -p <pkg> -d <dir> <schema files>` where:

- `-p <pkg>` specifies the target package
- `-d <dir>` specifies the directory to store generated files.

Schema Compiler Class

Property: jaxb.xsd_compiler.run.compilerWrapperClass

You must provide a custom class to invoke your compiler. This class should implement the `com.sun.jaxb_tck.lib.SchemaCompilerTool` *interface.*

When you run schema compilation through the JavaTest agent, your schema compiler will be executed in the agent's JVM. Your implementation must provide a custom class to invoke your schema compilation tool. This class should implement the `SchemaCompilerTool` interface:

```
package com.sun.jaxb_tck.lib;

public interface SchemaCompilerTool {

  /**
  * @param xsdFiles – array of strings containing schema files
  * @param packageName – the target package
  * @param outDir – output directory where java file(s) will be generated
  * @param out - output stream for logging
  * @param err - error stream for logging
  * @return 0 if java file(s) generated successfully
  */
  int compile(
    String[] xsdFiles,
    String packageName,
    File outDir,
    PrintStream out,
    PrintStream err
  );
}
```

Schema compilation should be accomplished during invocation of compile method. The XML Binding TCK includes the following class that is fully compatible with Eclipse's compatible implementation and that can be used for schema compilation: `com.sun.jaxb_tck.lib.SchemaCompiler`. The realization of the method `compile(String[] xsdFiles, String packageName, File outDir, PrintStream out, PrintStream err)` compiles a schema into `java` sources. If your implementation doesn't provide such a class, then create it and specify it here.

Schema Generator Presence

Property: jck.env.jaxb.schemagen.skipJ2XOptional

Does your implementation provide schema generator which is able to generate schemas from Java sources?

Java Architecture for XML Binding 3.0 specification does not require any implementation, which provides a possibility to generate the schemas from the java sources. If they are provided, then all specification requirements should be met.

Schema Generator Run Command

Property: jck.env.jaxb.schemagen.run.jxcCmd

Enter the command to run the schema generator.

This is the command used to invoke your schema generator. For example, for the Eclipse XML Binding implementation on Linux we provide a script `$TS_HOME/linux/bin/schemagen.sh`, which generates the schemas from the java files. The command is as follows:

```
/bin/sh /workspace/XMLB-TCK-3.0/linux/bin/schemagen.sh
```

For the Eclipse XML Binding implementation on Windows, we provide a script `%TS_HOME%\win32\bin\schemagen.bat`, which compiles the schemas into class files. The command is as follows:

```
c:\workspace\XMLB-TCK-3.0\win32\bin\schemagen.bat
```

The schemagen script supports the following mandatory command-line options: `schemagen -d <dir> <java files>`, where `-d <dir>` specifies the directory to store the generated files.

Schema Generator Class

Property: jck.env.jaxb.schemagen.run.schemagenWrapperClass

You must provide a custom class to invoke your schema generator. This class should implement the com.sun.jaxb_tck.lib.SchemaGenTool interface.

When you run schema generation through JavaTest agent, your schema generator will be executed in the agent's JVM. Your implementation must provide a custom class to invoke your schema generation tool. This class should implement the SchemaGenTool interface:

```
package com.sun.jaxb_tck.lib;

public interface SchemaGenTool {
  /**
  * @param javaFiles - array of strings containing java files
  * @param outDir - output directory where schema file(s) will be generated
  * @param out – output stream for logging
  * @param err – error stream for logging
  * @return 0 if schema file(s) generated successfully
  */
  int generate(
    String[] javaFiles,
    File outDir,
    PrintStream out,
    PrintStream err
  );
}
```

Schema generations should be complete during the invocation of the generate method. The XML Binding TCK includes the following class that is fully compatible with Eclipse's compatible implementation and that can be used for schema generation: `com.sun.jaxb_tck.lib.SchemaGen`. The realization of the method `generate(String[] javaFiles, File outDir, PrintStream out, PrintStream err)` generates the XML schemas from the `java` files. If your implementation doesn't provide such a class, then create it and specify it here.

Specify an Exclude List?

Property: jck.excludeList.needExcludeList

Do you wish to specify an exclude list?

Answer `Yes` if you want to specify an exclude list for the test run. If you select Yes, then select the location of the exclude list to use.

Answer `No` if you want to run all tests that are not filtered out by other run filters.

Which Exclude List?

Property: jck.excludeList.excludeListType

Which exclude list do you wish to use?

This question appears only if you answered Yes to the previous configuration interview question.

Select either the Initial option to use the exclude list provided with the test suite or the Specify option to specify the exclude list you wish to use.

Specify Exclude List Files

Property: jck.excludeList.customFiles

Specify the files that make up the exclude list you wish to use.

Click **Add** to activate a file chooser with which you select an exclude list file. Exclude list files conventionally use the extension `.jtx`.

Specify Keywords?

Property: jck.keywords.needKeywords

Do you wish to specify a keyword expression to select/reject tests based on keywords contained in each test description?

Keywords are tokens associated with specific tests.

Answer `Yes` to run tests based on the keyword and the other run filter settings. If you select Yes, then you are asked to specify the keyword to use.

Answer `No` to run all tests that are not filtered out by other run filters.

Specify Status?

Property: jck.priorStatus.needStatus

Do you wish to select tests to run based on their result in a previous run?

Answer Yes to run tests based on the status from the previous test run and the other run filter settings. If you select Yes, then you are asked for the status or statuses.

Answer No to run all tests that are not filtered out by other run filters.

Concurrency

Property: jck.concurrency.concurrency

The JavaTest harness can run tests concurrently. Specify the maximum number of tests that should be run concurrently.

The JavaTest harness can run tests concurrently. If you are running the tests on a multiprocessor computer, then the concurrency can accelerate your test runs. The default range of values that the JavaTest harness uses is from 1 to 50. For the first test run, keep this value set to 1.

Time Factor

Property: jck.timeout.timeout

Specify a time factor that is applied to each test's default timeout. For example, specifying 2 doubles the time for each test (the default is 1).

The default 1.0 corresponds to 10 minutes. Use the floating point format specific to your locale.

Setting a proper time factor value requires consideration of such technical factors as the network connector speed.

4.2 Using the JavaTest Harness Software

To run the XML Binding TCK test suite using the JavaTest harness software in JavaTest batch mode, from the command line in your shell environment, please proceed directly to Chapter 5, "Executing Tests."

5 Executing Tests

The XML Binding TCK uses the JavaTest harness to execute the tests in the test suite. For detailed instructions that explain how to run and use JavaTest, see the JavaTest User's Guide and Reference in the documentation bundle.

This chapter includes the following topics:

- Starting JavaTest
- Running a Subset of the Tests
- Running the TCK Against your selected CI
- Running the TCK Against a Vendor's Implementation
- Test Reports

The instructions in this chapter assume that you have installed and configured your test environment as described in Chapter 3, "Installation," and Chapter 4, "Setup and Configuration,", respectively.

5.1 Starting JavaTest

General way to run the XML Binding TCK using the JavaTest harness software is from the command line in your shell environment

5.1.1 To Start JavaTest in Command-Line Mode

1. Change to `$TS_HOME/lib` directory.
2. Start JavaTest using the following command:

```
${JAVA_HOME}/bin/java -jar ${TS_HOME}/lib/javatest.jar \
  -batch -testsuite ${TS_HOME} -open ${TS_HOME}/lib/javasoft-multiJVM.jti \
  -workdir -create ${TS_HOME}/work -runtests
```

Example 5-1 XML Binding TCK Signature Tests

To run the XML Binding TCK signature tests, enter the following commands:

```
cd ${TS_HOME}/tests/api/signaturetest/jaxb
ant runclient
```

Example 5-2 Single Test Directory

To run a single test directory, enter the following commands:

```
cd ${TS_HOME}/tests/api/jakarta_xml/bind/JAXBContext
ant runclient
```

Example 5-3 Subset of Test Directories

To run a subset of test directories, enter the following commands:

```
cd ${TS_HOME}/tests/api/jakarta_xml/bind
ant runclient
```

5.2 Running a Subset of the Tests

Use the following modes to run a subset of the tests:

- Section 5.2.1, "To Run a Subset of Tests in Command-Line Mode"
- Section 5.2.2, "To Run a Subset of Tests in Batch Mode Based on Prior Result Status"

5.2.1 To Run a Subset of Tests in Command-Line Mode

1. Change to the directory containing the tests you want to run.
2. Start the test run by executing the following command:

   ```
   ant runclient
   ```

   The tests in the directory and its subdirectories are run.

5.2.2 To Run a Subset of Tests in Batch Mode Based on Prior Result Status

You can run certain tests in batch mode based on the test's prior run status by specifying the `priorStatus` system property when invoking `ant`

Invoke `ant` with the `priorStatus` property.

The accepted values for the `priorStatus` property are any combination of the following:

- `fail`
- `pass`
- `error`
- `notRun`

For example, you could run all the XML Binding tests with a status of failed and error by invoking the following commands:

```
ant -DpriorStatus="fail,error" runclient
```

Note that multiple `priorStatus` values must be separated by commas.

5.3 Running the TCK Against another CI

Some test scenarios are designed to ensure that the configuration and deployment of all the prebuilt XML Binding TCK tests against one Compatible Implementation are successful operating with other compatible implementations, and that the TCK is ready for compatibility testing against the Vendor and Compatible Implementations.

1. Verify that you have followed the configuration instructions in Section 4.1, "Configuring Your Environment to Run the TCK Against the Compatible Implementation."
2. Run the tests, as described in Section 5.1, "Starting JavaTest," and, if desired, Section 5.2, "Running a Subset of the Tests."

5.4 Running the TCK Against a Vendor's Implementation

This test scenario is one of the compatibility test phases that all Vendors must pass.

1. Verify that you have followed the configuration instructions in Section 4.2, "Configuring Your Environment to Repackage and Run the TCK Against the Vendor Implementation."

2. Run the tests, as described in Section 5.1, "Starting JavaTest," and, if desired, Section 5.2, "Running a Subset of the Tests."

5.5 Test Reports

A set of report files is created for every test run. These report files can be found in the report directory you specify. After a test run is completed, the JavaTest harness writes HTML reports for the test run. You can view these files in the JavaTest ReportBrowser when running in GUI mode, or in the web browser of your choice outside the JavaTest interface.

To see all of the HTML report files, enter the URL of the `report.html` file. This file is the root file that links to all of the other HTML reports.

The JavaTest harness also creates a `summary.txt` file in the report directory that you can open in any text editor. The `summary.txt` file contains a list of all tests that were run, their test results, and their status messages.

5.5.1 Creating Test Reports

To generate test reports, enter the following command:

```
${JAVA_HOME}/bin/java -jar javatest.jar \
  -workdir ${TS_HOME}/work -writereport ${TS_HOME}/report
```

Choose proper directories for XML Binding TCK tests working directory and report directory.

5.5.2 Viewing an Existing Test Report

Use the Web browser of your choice to view the `index.html` file in the report directory you specified from the command line.

6 Debugging Test Problems

There are a number of reasons that tests can fail to execute properly. This chapter provides some approaches for dealing with these failures. Please note that most of these suggestions are only relevant when running the test harness in GUI mode.

This chapter includes the following topics:

6.1 Overview

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

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

The Test Manager GUI provides you with a number of tools for effectively troubleshooting a test run. See the JavaTest User's Guide and JavaTest online help for detailed descriptions of the tools described in this chapter. Ant test execution tasks provide command-line users with immediate test execution feedback to the display. Available JTR report files and log files can also help command-line users troubleshoot test run problems.

For every test run, the JavaTest harness creates a set of report files in the reports directory, which you specified by setting the `report.dir` property in the `${TS_HOME}/lib/javasoft-multiJVM.jti` file. The report files contain information about the test description, environment, messages, properties used by the test, status of the test, and test result. After a test run is completed, the JavaTest harness writes HTML reports for the test run. You can view these files in the JavaTest ReportBrowser when running in GUI mode, or in the Web browser of your choice outside the JavaTest interface. To see all of the HTML report files, enter the URL of the `report.html` file. This file is the root file that links to all of the other HTML reports.

The JavaTest harness also creates a `summary.txt` file in the report directory that you can open in any

text editor. The `summary.txt` file contains a list of all tests that were run, their test results, and their status messages.

The work directory, which you specified by setting the `work.dir` property in the `${TS_HOME}/lib/javasoft-multiJVM.jti` file, contains several files that were deposited there during test execution: `harness.trace`, `log.txt`, `lastRun.txt`, and `testsuite`. Most of these files provide information about the harness and environment in which the tests were executed.

You can set `harness.log.traceflag=true` in `${TS_HOME}/lib/javasoft-multiJVM.jti` to get more debugging information.

If a large number of tests failed, you should read Configuration Failures to see if a configuration issue is the cause of the failures.

6.2 Test Tree

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

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

6.3 Folder Information

Click a folder in the test tree in the JavaTest GUI to display its tabs.

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

6.4 Test Information

To display information about a test in the JavaTest GUI, click its icon in the test tree or double-click its name in a folder status tab. The tab contains detailed information about the test run and, at the bottom of the window, a brief status message identifying the type of failure or error. This message may be

sufficient for you to identify the cause of the error or failure.

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

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

You can set `harness.log.traceflag=true` in `${TS_HOME}/lib/javasoft-multiJVM.jti` to get more debugging information.

6.5 Report Files

Report files are another good source of troubleshooting information. You may view the individual test results of a batch run in the JavaTest Summary window, but there are also a wide range of HTML report files that you can view in the JavaTest ReportBrowser or in the external browser or your choice following a test run. See Section 5.5, "Test Reports," for more information.

6.6 Configuration Failures

Configuration failures are easily recognized because many tests fail the same way. When all your tests begin to fail, you may want to stop the run immediately and start viewing individual test output. However, in the case of full-scale launching problems where no tests are actually processed, report files are usually not created (though sometimes a small `harness.trace` file in the report directory is written).

Here are few tips to verify the errors:

1. Check your `.jti` file settings in the Configuration Editor. You may have launched the JavaTest harness with prior status set to `any` instead of `ignore`.
2. Check if the `env.html` report file is generated, else, the JavaTest GUI provides the ability to view evaluated variables used by the tests. Select **Configure** and then **Show Test Environment** from the Test manager menu to view the contents of the test environment.
3. Verify if the Configuration Editor has generated a configuration question log when you complete a configuration interview. You can use the configuration question log to review your answers to all the questions in the current configuration. Select **Configure** and **Show Question Log** from the **Test**

Manager menu to view the current configuration interview.

6.7 Test Manager Properties

You can view the properties of a test manager by selecting **View** and then **Properties** from the **Test Manager** menu. The Test Manager Properties dialog box contains Test Suite, Work Directory, Configuration, and Plugin information.

See JavaTest User's Guide , Graphical User Interface, or JavaTest online help for a detailed description of the Test Manager Properties dialog box.

6.8 Test Suite Errors

If the JavaTest harness detects test suite errors, then it displays an advisory dialog box. You can view detailed information about the test suite errors by selecting View and then **Test Suite Errors** from the Test Manager menu.

See JavaTest User's Guide , Graphical User Interface, or JavaTest online help for a detailed description of the **Test Manager: Test Suite Errors** dialog box.

6.9 How Tests are Executed

To better understand and debug the test failures, this section describes how the tests are selected for a test run and how they are then executed.

Test Results

Test Execution results are reported as one of the three states:

- Pass: A test passes when the functionality being tested behaves as expected. All tests are expected to pass.
- Fail: A test fails when the functionality being tested does not behave as expected.
- Error: A test is considered to cause error when something (usually a configuration problem) keeps the test from being executed as expected. Errors often indicate a systemic problem - a single configuration problem can cause many tests to fail. For example, if the path to the Java runtime is configured incorrectly, then no tests can run and will result in an error.

How Tests Are Selected For a Test Run

Prior to the start of a test run, the JavaTest harness selects tests for the run based on the following factors:

- Tests to be Run: The JavaTest harness finds tests listed in the Tests to be Run field of the JavaTest configuration. You can specify the sub-branches of the tree as a way of limiting the tests that are executed during a test run. The JavaTest harness browses through the tree starting with its sub-branches or tests you specify, and executes all tests that it finds.
- Exclude List: Tests listed in the appropriate exclude list are deselected prior to the start of a test run. For details about exclude lists and their role in the certification process, see Procedure for Certification.
- Keywords: A test can be selected based on keywords specified in the Keywords field in the test description. The possible keywords are the following: schema, document, runtime, positive, negative, bindinfo, jaxb_not_required, java_to_schema.
- Prior Status: Use the combo box and check boxes to select tests in a test run based on their outcome on a prior test run. Prior status is evaluated on a test-bytest basis using information stored in result files (.jtr) written in the work directory.

Test Execution

Based on the test keywords and the `executeClass` parameter of the test description, all the tests are divided in four execution models:

- Schema Tests: The keyword schema indicates this test execution model. The schema tests are the Schema Compiler Tests. These tests have a schema file (*.xsd) in the test description parameter source. The schema is compiled to the work directory, and the output directory option of the XML Binding schema compiler is used.
  The package specified in the test description parameter package is used to set the output package parameter to the schema compiler. If the test has the keyword negative, then the schema compiler must report an error compiling the test schema. Otherwise, the schema is positive and may have XML Binding custom binding information if the keyword bindinfo is present. + The positive schema is expected to compile successfully. After the compilation passes, the signature test (the class specified in the test description parameter executeClass) runs with arguments specified in the test description parameter executeArgs.
- Document Tests: The keyword document along with the class JAXBTest specified in the test description parameter executeClass indicate this test execution model. The document tests are the subset of the Content Tree Tests, with the exception of the content tree modification stage.
  The last two stages are applicable only to valid XML documents. The first stage (schema compilation) is performed as for the schema tests. All stages except the first one are performed by executing appropriate test cases (unmarshal, compareContent, and marshal respectively) of the class specified in test description parameter executeClass, with arguments specified in the test description parameter executeArgs.
  All parameter entries starting with *$* are resolved in the environment, but no errors occur if any of

these values are not defined. If the exclude list identifies test cases to be excluded, then these cases are added to the test arguments using the -exclude option. Note that all schemas used in the document tests are valid and the tests may not have the keyword negative. Invalid documents are indicated by option -invalid in the parameter executeClass.

- XML Binding Framework API Tests: The XML Binding framework API tests have their own classes (other than the signature test or JAXBTest) specified in the test description parameter executeClass. The tests may or may not have schema file specified in parameter schema.
  First, if the schema is specified, it is compiled as described for the schema tests. Then the test executes the class specified in test description parameter executeClass with arguments specified in the test description parameter executeArgs.
  All parameter entries starting with *$* are resolved in the environment, but no errors occur if any of these values are not defined. If the exclude list identifies test cases to be excluded, then these cases are added to the test arguments using the -exclude option.
- Java -to-Schema Tests: The keyword /java_to_schema/ indicates this test execution model. The Java-to-Schema Tests are the same as described in Schema Compiler Tests. These tests have a Java files (*.java) in the test description parameter source. The generated schema is placed in the work directory, and the output directory option of the XML Binding schema generator is used. If the test has the attribute negative, then the schema generator must report an error processing the test Java files. Otherwise, the Java code is correctly annotated and should be processed successfully. After the generation passes, the validation test (the class specified in the test description parameter /executeClass/) runs with arguments specified in the test description parameter /executeArgs/. The validation test checks that a valid XML document, taken from the test description parameter /executeArgs/ follows the just-generated schema, and fails to check invalid XML documents.
- XML Validation Tests: XML Validation can be implemented through JAXP or some other mechanism. If the validation is implemented through JAXP, then validation checking tests can be skipped by deselecting them in the test interview. If you use some other tools for schema validation, then XML validation tests are mandatory.

A Frequently Asked Questions

This appendix contains the following questions.

A.1 Where do I start to debug a test failure?

From the JavaTest GUI, you can view recently run tests using the Test Results Summary, by selecting the red Failed tab or the blue Error tab. See Chapter 6, "Debugging Test Problems," for more information.

A.2 How do I restart a crashed test run?

If you need to restart a test run, you can figure out which test crashed the test suite by looking at the `harness.trace` file. The `harness.trace` file is in the report directory that you supplied to the JavaTest GUI or parameter file. Examine this trace file, then change the JavaTest GUI initial files to that location or to a directory location below that file, and restart. This will overwrite only `.jtr` files that you rerun. As long as you do not change the value of the GUI work directory, you can continue testing and then later compile a complete report to include results from all such partial runs.

A.3 What would cause tests be added to the exclude list?

The JavaTest exclude file (`${TS_HOME}/lib/jaxb_tck30.jtx`) contains all tests that are not required to be run. The following is a list of reasons for a test to be included in the Exclude List:

- An error in a compatible implementation that does not allow the test to execute properly has been discovered.
- An error in the specification that was used as the basis of the test has been discovered.
- An error in the test has been discovered.