Technology Compatibility Kit Reference Guide for JSR 346: Contexts and Dependency Injection for Java EE 1.1

Specification Lead: Red Hat Inc.

Pete Muir
JSR 346: Contexts and Dependency Injection (CDI) for Java EE 1.1 specification lead

Gavin King
JSR 299: Contexts and Dependency Injection (CDI) for Java EE 1.0 specification lead

Martin Kouba
CDI TCK lead

Dan Allen
CDI TCK developer

Preface

This guide describes how to download, install, configure, and run the Technology Compatibility Kit (TCK) used to verify the compatibility of an implementation of the JSR 346: Context and Dependency Injection for Java EE (CDI) 1.1 specification.

The CDI TCK is built atop TestNG framework and Arquillian platform. The CDI TCK uses the Arquillian version *1.0.3.Final* to execute the test suite.

The CDI TCK is provided under the *Apache Public License 2.0* [http://www.apache.org/licenses/LICENSE-2.0].

1. Who Should Use This Book

This guide is for implementors of the Context and Dependency Injection for Java EE 1.1 technology to assist in running the test suite that verifies the compatibility of their implementation.

2. Before You Read This Book

Before reading this guide, you should familiarize yourself with the Java EE programming model, specifically the Enterprise JavaBeans (EJB) 3.1 and the Contexts and Dependency Injection for Java EE 1.1 specifications. A good resource for the Java EE programming model is the *JCP* [http://jcp.org] web site.

The CDI TCK is based on the Context and Dependency Injection for Java EE 1.1 technology specification (JSR 346). Information about the specification, including links to the specification documents, can be found on the *JSR 346 JCP page* [http://jcp.org/en/jsr/detail?id=346].

Before running the tests in the CDI TCK, read and become familiar with the Arquillian testing platform. A good starting point could be a series of *Arquillian Guides* [http://arquillian.org/guides/].

3. How This Book Is Organized

If you are running the CDI TCK for the first time, read *Chapter 1, Introduction (CDI TCK)* completely for the necessary background information about the TCK. Once you have reviewed that material, perform the steps outlined in the remaining chapters.

- *Chapter 1, Introduction (CDI TCK)* gives an overview of the principles that apply generally to all Technology Compatibility Kits (TCKs), outlines the appeals process and describes the CDI TCK architecture and components. It also includes a broad overview of how the TCK is executed and lists the platforms on which the TCK has been tested and verified.
- *Chapter 2, Appeals Process* explains the process to be followed by an implementor should they wish to challenge any test in the TCK.
- *Chapter 3, Installation* explains where to obtain the required software for the CDI TCK and how to install it. It covers both the primary TCK components as well as tools useful for troubleshooting tests.

- *Chapter 4, Configuration* details the configuration of the JBoss Test Harness, how to create a TCK runner for the TCK test suite and the mechanics of how an in-container test is conducted.
- *Chapter 5, Reporting* explains the test reports that are generated by the TCK test suite and introduces the TCK audit report as a tool for measuring the completeness of the TCK in testing the JSR 346 specification and in understanding how testcases relate to the specification.
- *Chapter 7, Executing the Test Suite* documents how the TCK test suite is executed. It covers both modes supported by the TCK, standalone and in-container, and shows how to dump the generated test artifacts to disk.
- *Chapter 8, Running Tests in Eclipse* shows how to run individual tests in Eclipse and advises the best way to setup your Eclipse workspace for running the tests.
- *Chapter 9, Debugging Tests in Eclipse* builds on *Chapter 8, Running Tests in Eclipse* by detailing how to debug individual tests in Eclipse.

Part I. Getting Acquainted with the TCK

The CDI TCK must be used to ensure that your implementation conforms to the CDI specification. This part introduces the TCK, gives some background about its purpose, states the requirements for passing the TCK and outlines the appeals process.

In this part you will learn where to obtain the CDI TCK and supporting software. You are then presented with recommendations of how to organize and configure the software so that you are ready to execute the TCK.

Finally, it discusses the reporting provided by the TCK.

Introduction (CDI TCK)

This chapter explains the purpose of a TCK and identifies the foundation elements of the CDI TCK.

1.1. TCK Primer

A TCK, or Technology Compatibility Kit, is one of the three required pieces for any JSR (the other two being the specification document and the reference implementation). The TCK is a set of tools and tests to verify that an implementation of the technology conforms to the specification. The tests are the primary component, but the tools serve an equally critical role of providing a framework and/or set of SPIs for executing the tests.

The tests in the TCK are derived from assertions in the written specification document. The assertions are itemized in an XML document, where they each get assigned a unique identifier, and materialize as a suite of automated tests that collectively validate whether an implementation complies with the aforementioned assertions, and in turn the specification. For a particular implementation to be certified, all of the required tests must pass (i.e., the provided test suite must be run unmodified).

A TCK is entirely implementation agnostic. Ideally, it should validate assertions by consulting the specification's public API. However, when the information returned by the public API is not low-level enough to validate the assertion, the implementation must be consulted directly. In this case, the TCK provides an independent API as part of a porting package that enables this transparency. The porting package must be implemented for each CDI implementation. *Section 1.3.2, "CDI TCK Components"* introduces the porting package and *Section 4.3, "The Porting Package"* covers the requirements for implementing it.

Note

Oracle Corporation will implement the porting package for the CDI RI and test the CDI RI on the Java EE Reference Implementation.

1.2. Compatibility Testing

The goal of any specification is to eliminate portability problems so long as the program which uses the implementation also conforms to the rules laid out in the specification.

Executing the TCK is a form of compatibility testing. It's important to understand that compatibility testing is distinctly different from product testing. The TCK is not concerned with robustness, performance or ease of use, and therefore cannot vouch for how well an implementation meets these criteria. What a TCK can do is to ensure the exactness of an implementation as it relates to the specification.

Compatibility testing of any feature relies on both a complete specification and a complete reference implementation. The reference implementation demonstrates how each test can

be passed and provides additional context to the implementor during development for the corresponding assertion.

1.2.1. Why Compatibility Is Important

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

- Compatibility testing is the means by which the JCP ensures that the Java platform does not become fragmented as it's ported to different operating systems and hardware.
- Compatibility testing benefits developers working in the Java programming language, enabling them to write applications once and deploy them across heterogeneous computing environments without porting.
- Compatibility testing enables application users to obtain applications from disparate sources and deploy them with confidence.
- Conformance testing benefits Java platform implementors by ensuring the same extent of reliability for all Java platform ports.

The CDI specification goes to great lengths to ensure that programs written for Java EE are compatible and the TCK is rigorous about enforcing the rules the specification lays down.

1.3. About the CDI TCK

The CDI TCK is designed as a portable, configurable and automated test suite for verifying the compatibility of an implementation of the JSR 346: Contexts and Dependency Injection for Java EE 1.1 specification. The test suite is built atop TestNG framework and Arquillian platform.

Each test class in the suite acts as a deployable unit. The deployable units, or artifacts, can be either a WAR or an EAR.

Note

The test archives are built with ShrinkWrap, a Java API for creating archives. ShrinkWrap is a part of the Arqullian platform ecosystem.

1.3.1. CDI TCK Specifications and Requirements

This section lists the applicable requirements and specifications for the CDI TCK.

- **Specification requirements** - Software requirements for a CDI implementation are itemized in section 1.2, "Relationship to other specifications" in the CDI specification, with details provided throughout the specification. Generally, the CDI specification targets the Java EE 7 platform and is aligned with its specifications.

- **Contexts and Dependency Injection for Java EE 1.1 API** - The Java API defined in the CDI specification and provided by the reference implementation.
- **Testing platform** - The CDI TCK requires version 1.0.2.Final of the Arquillian (*http://arquillian.org*). The TCK test suite is based on TestNG 6.x (*http://testng.org*). .
- **Porting Package** - An implementation of SPIs that are required for the test suite to run the in-container tests and at times extend the CDI 1.1 API to provide extra information to the TCK.
- **TCK Audit Tool** - An itemization of the assertions in the specification documents which are cross referenced by the individual tests. Describes how well the TCK covers the specification.
- **Reference runtime** - The designated reference runtimes for compatibility testing of the CDI specification is the Oracle Java Platform, Enterprise Edition (Java EE) 7 reference implementation (RI).
- **JSR 330** - CDI builds on JSR 330, and as such JSR 346 implementations must additionally pass the JSR 330 TCK.

Tip

The TCK distribution includes `weld/porting-package-lib/weld-inject-tck-runner-X.Y.Z-Q-tests.jar` which contains two classes showing how the CDI RI passes the JSR 330 TCK. The source for these classes is available from *https://github.com/weld/core/tree/2.0/inject-tck-runner/src/test/java/org/jboss/weld/atinject/tck*

1.3.2. CDI TCK Components

The CDI TCK includes the following components:

- **Arquillian 1.0.3.Final**
- **TestNG 6.3**
- **Porting Package SPIs** - Extensions to the CDI SPIs to allow testing of a container.
- **The test suite**, which is a collection of TestNG tests, the TestNG test suite descriptor and supplemental resources that configure CDI and other software components.
- **The TCK audit** is used to list out the assertions identified in the CDI specification. It matches the assertions to testcases in the test suite by unique identifier and produces a coverage report.

  The audit document is provided along with the TCK; at least 95% of assertions are tested. Each assertion is defined with a reference to a chapter, section and paragraph from the specification document, making it easy for the implementor to locate the language in the specification document that supports the feature being tested.

- **TCK documentation** accompanied by release notes identifying updates between versions.

Note

Oracle Corporation will implement the porting package for the CDI RI and test the CDI RI on the Java EE Reference Implementation .

The CDI TCK has been tested run on following platforms:

- WildFly 8.x using Oracle Java SE 7 on Red Hat Enterprise Linux 5.2

CDI supports Java EE 5, Java EE 6, Java EE 6 Web Profile, Java EE 7, Java EE 7 Web Profile, Embeddable EJB 3.1, and the Embeddable EJB 3.2. The TCK will execute on any of these runtimes, but is only part of the CTS for Java EE 7 and Java EE 7 Web Profile.

Appeals Process

While the CDI TCK is rigorous about enforcing an implementation's conformance to the JSR 346 specification, it's reasonable to assume that an implementor may discover new and/or better ways to validate the assertions. This chapter covers the appeals process, defined by the Specification Lead, Red Hat Middleware LLC., which allows implementors of the JSR 346 specification to challenge one or more tests defined by the CDI TCK.

The appeals process identifies who can make challenges to the TCK, what challenges to the TCK may be submitted, how these challenges are submitted, how and by whom challenges are addressed and how accepted challenges to the TCK are managed.

Following the recent adoption of transparency in the JCP, implementors are encouraged to make their appeals public, which this process facilitates. The JCP community should recognize that issue reports are a central aspect of any good software and it's only natural to point out shortcomings and strive to make improvements. Despite this good faith, not all implementors will be comfortable with a public appeals process. Instructions about how to make a private appeal are therefore provided.

2.1. Who can make challenges to the TCK?

Any implementor may submit an appeal to challenge one or more tests in the CDI TCK. In fact, members of the JSR 346 Expert Group (EG) encourage this level of participation.

2.2. What challenges to the TCK may be submitted?

Any test case (e.g., test class, `@Test` method), test case configuration (e.g., beans.xml), test beans, annotations and other resources may be challenged by an appeal.

What is generally not challengeable are the assertions made by the specification. The specification document is controlled by a separate process and challenges to it should be handled through the JSR 346 EG by sending an e-mail to *jsr346-comments@jcp.org* [mailto:jsr346-comments@jcp.org].

2.3. How these challenges are submitted?

To submit a challenge, a new issue should be created in the *CDI TCK project* [https://jira.jboss.org/jira/browse/CDITCK] of the JBoss JIRA using the Issue Type: Bug. The appellant should complete the Summary, Component (TCK Appeal), Environment and Description Field only. Any communication regarding the issue should be pursed in the comments of the filed issue for accurate record.

To submit an issue in the JBoss JIRA, you must have a (free) JBoss.org member account. You can create a member account using the *on-line registration* [https://community.jboss.org/register.jspa].

If you wish to make a private challenge, you should follow the above procedure, setting the Security Level to Private. Only the issue reporter, TCK Project Lead and designates will be able to view the issue.

2.4. How and by whom challenges are addressed?

The challenges will be addressed in a timely fashion by the CDI TCK Project Lead, as designated by Specification Lead, Red Hat Middleware LLC. or his/her designate. The appellant can also monitor the process by following the issue report filed in the *CDI TCK project* [https://jira.jboss.org/jira/browse/CDITCK] of the JBoss JIRA.

The current TCK Project Lead is listed on the *CDI TCK Project Summary Page* [https://jira.jboss.org/jira/browse/CDITCK] on the JBoss JIRA.

2.5. How accepted challenges to the TCK are managed?

Accepted challenges will be acknowledged via the filed issue's comment section. Communication between the CDI TCK Project Lead and the appellant will take place via the issue comments. The issue's status will be set to "Resolved" when the TCK project lead believes the issue to be resolved. The appellant should, within 30 days, either close the issue if they agree, or reopen the issue if they do not believe the issue to be resolved.

Resolved issue not addressed for 30 days will be closed by the TCK Project Lead. If the TCK Project Lead and appellant are unable to agree on the issue resolution, it will be referred to the JSR 346 specification lead or his/her designate.

Periodically, an updated TCK will be released, containing tests altered due to challenges - no new tests will be added. Implementations are required to pass the updated TCK. This release stream is named 1.1.x, where x will be incremented.

Additionally, new tests will be added to the TCK improving coverage of the specification. We encourage implementations to pass this TCK, however it is not required. This release stream is named 1.y.z where y > 1.

Installation

This chapter explains how to obtain the TCK and supporting software and provides recommendations for how to install/extract it on your system.

3.1. Obtaining the Software

You can obtain a release of the CDI TCK project from the *download page* [http://http://www.cdi-spec.org/download/] on the CDI specification website. The release stream for JSR 346 is named *1.1.x*. The CDI TCK is distributed as a ZIP file, which contains the TCK artifacts (the test suite binary and source, porting package API binary and source, the test suite configuration file, the audit source and report) in `/artifacts` and documentation in `/doc`. The TCK library dependencies are not part of the distribution and can be downloaded on demand (see readme.txt file in `/lib`).

You can also download the current source code from *GitHub repository* [https://github.com/jboss/cdi-tck].

The TCK project is also available in the Maven Central repository. The POM file defines all dependencies required to build the TCK.

```
<dependency>
   <groupId>org.jboss.cdi.tck</groupId>
   <artifactId>cdi-tck-impl</artifactId>
   <version>${cdi-tck.version}</version>
</dependency>
```

Executing the TCK requires a Java EE 7 or better runtime environment (i.e., application server), to which the test artifacts are deployed and the individual tests are invoked. The TCK does not depend on any particular Java EE implementation.

The JSR 346: Contexts and Dependency Injection for Java EE 1.1 reference implementation (RI) project is named Weld. The release stream for JSR 346 is named *2.x*. You can obtain the latest release from the *download page* [http://weld.cdi-spec.org/download/] on the Weld website.

Note

Weld is not required for running the CDI TCK, but it can be used as a reference for familiarizing yourself with the TCK before testing your own CDI implementation.

Naturally, to execute Java programs, you must have a Java SE runtime environment. The TCK requires Java 7 or better, which you can obtain from the *Java Software* [http://www.oracle.com/technetwork/java/index.html] website.

3.2. The TCK Environment

The TCK requires the following two Java runtime environments:

- Java 7 or better
- Java EE 7 or better (e.g., WildFly 8.x or GlassFish V4)

You should refer to vendor instructions for how to install the runtime environment.

The rest of the TCK software can simply be extracted. It's recommended that you create a folder named *jsr346* to hold all of the jsr346-related projects. Then, extract the TCK distribution into a subfolder named *tck*. If you have downloaded the Weld distribution, extract it into a sibling folder named *weld*. The resulting folder structure is shown here:

Note

This layout is assumed through all descriptions in this reference guide.

```
jsr346/
  weld/
  tck/
```

Each test class is treated as an individual artifact. All test methods (i.e., methods annotated with `@Test`) in the test class are run in the application, meaning bean discovery occurs exactly once per artifact and the same BeanManager is used by each test method in the class.

Running the TCK against the CDI RI (Weld) and WildFly

- First, you should download WildFly 8.x from the WildFly *project page* [http://www.wildfly.org/download/].
- Set the JBOSS_HOME environment variable to the location of the WildFly software.

The CDI TCK distribution includes a TCK runner that executes the TCK using Weld as the CDI implementation and WildFly as the Java EE runtime. To run the TCK:

- You need to install Maven. You can find documentation on how to install Maven in the *Maven: The Definitive Guide* [http://www.sonatype.com/books/

maven-book/reference/installation-sect-maven-install.html] book published by Sonatype.

- Next, instruct Maven to run the TCK:

```
cd jsr346/tck/weld/jboss-tck-runner
mvn test -Dincontainer
```

- Use `cdi.tck.version` system property to specify particular TCK version:

```
mvn test -Dincontainer -Dcdi.tck.version=1.1.2.Final
```

- TestNG will report, via Maven, the outcome of the run, and report any failures on the console. Details can be found in `target/surefire-reports/TestSuite.txt`.

3.3. Eclipse Plugins

Eclipse, or any other IDE, is not required to execute or pass the TCK. However an implementor may wish to execute tests in an IDE to aid debugging the tests. This section introduces two essential Eclipse plugins, TestNG and Maven, and points you to resources explaining how to install them.

3.3.1. TestNG Plugin

The TCK test suite is built on the TestNG. Therefore, having the TestNG plugin installed in Eclipse is essential. Instructions for using the TestNG update site to add the TestNG plugin to Eclipse are provided on the TestNG *download page* [http://testng.org/doc/download.html]. You can find a tutorial that explains how to use the TestNG plugin on the TestNG *Eclipse page* [http://testng.org/doc/eclipse.html].

3.3.2. Maven Plugin (m2e)

Another useful plugin is m2e. The TCK project uses Maven. Therefore, to work with TCK in Eclipse, you may wish to have native support for Maven projects, which the m2e plugin provides. Instructions for using the m2e update site to add the m2e plugin to Eclipse are provided on the m2e *home page* [http://eclipse.org/m2e/].

You can alternatively use the Eclipse plugin for Maven to generate native Eclipse projects from Maven projects.

If you have Maven installed, you have everything you need. Just execute the following command from any Maven project to produce the Eclipse project files.

```
mvn eclipse:eclipse
```

Again, the Eclipse plugins are not required to execute the TCK, but can be very helpful when validating an implementation against the TCK test suite and especially when using the modules from the project.

Configuration

This chapter lays out how to configure the TCK Harness by specifying the SPI implementation classes, defining the target container connection information, and various other switches. You then learn how to setup a TCK runner project that executes the TCK test suite, putting these settings into practice.

4.1. TCK Properties

System properties and/or the resource `META-INF/cdi-tck.properties`, a Java properties file, are used to configure the TCK.

You should set the following required properties:

Table 4.1. Required TCK Configuration Properties

Property = Example Value	Description
`org.jboss.cdi.tck.libraryDirectory=/path/to/extra/libraries`	The directory containing extra JARs to be placed in the test archive library directory such as the porting package implementation.
`org.jboss.cdi.tck.testDataSource=java:jboss/datasources/ExampleDS`	A few TCK tests work with Java EE persistence services (JPA, JTA) and require a data source to be provided. This property defines JNDI name of such resource. Required for the tests within the *persistence* test group.
`org.jboss.cdi.tck.testJmsConnectionFactory=java:/ConnectionFactory`	The JNDI name of the JMS test ConnectionFactory. Required for the tests within the *jms* test group.
`org.jboss.cdi.tck.testJmsQueue=java:/queue/test`	The JNDI name of the JMS test Queue. Required for the tests within the *jms* test group.

Property = Example Value	Description
`org.jboss.cdi.tck.testJmsTopic=java:/topic/test`	The JNDI name of the JMS test Topic. Required for the tests within the *jms* test group.

Table 4.2. Optional TCK Configuration Properties

Property = Example Value	Description
`org.jboss.cdi.tck.testTimeoutFactor=200`	Tests use this percentage value to adjust the final timeout (e.g. when waiting for some async processing) so that it's possible to configure timeouts according to the testing runtime performance and throughput. The value must be an integer greater than zero. The default value is 100% - i.e. timeouts will remain the same.

4.2. Arquillian settings

The Arquillian testing platform will look for configuration settings in a file named *arquillian.xml* in the root of your classpath. If it exists it will be auto loaded, else default values will be used. This file is not a requirement however it's very useful for container configuration. See an example configuration for JBoss TCK runner:

weld/jboss-tck-runner/src/test/jbossas7/arquillian.xml

4.3. The Porting Package

The CDI TCK relies on an implementation of the porting package to function. There are times when the tests need to tap directly into the CDI implementation to manipulate behavior or verify results. The porting package includes a set of SPIs that provide the TCK with this level of access without tying the tests to a given implementation.

The SPI classes in the CDI TCK are as follows:

- `org.jboss.cdi.tck.spi.Beans`
- `org.jboss.cdi.tck.spi.Contexts`
- `org.jboss.cdi.tck.spi.EL`

Please consult the JavaDoc for these interfaces for the implementation requirements.

4.4. Using the CDI TCK with the Java EE Web Profile

You can configure the CDI TCK to just run tests appropriate to the Java EE Web Profile by excluding TestNG group *javaee-full*, e.g. via maven-surefire-plugin configuration:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
       <excludedGroups>javaee-full</excludedGroups>
    </configuration>
 </plugin>
```

4.5. Configuring TestNG to execute the TCK

The CDI TCK is built atop Arquillian and TestNG, and it's TestNG that is responsible for selecting the tests to execute, the order of execution, and reporting the results. Detailed TestNG documentation can be found at *testng.org* [http://testng.org/doc/documentation-main.html].

Certain TestNG configuration file must be run by TestNG 6.3 (described by the TestNG documentation as "with a `testng.xml` file") unmodified for an implementation to pass the TCK. The TCK distribution contains the configuration file accurate at the date of the release - `artifacts/cdi-tck-impl-suite.xml`. However this artifact may not be up to date due to unresolved challenges or pending releases. Therefore a canonical configuration file exists. This file is located in the CDI TCK source code repository at *${CORRESPONDING_BRANCH_ROOT}/impl/src/main/resources/tck-tests.xml*.

Note

The canonical configuration file for CDI TCK 1.1.x is located at *https://github.com/cdi-spec/cdi-tck/blob/1.1/impl/src/main/resources/tck-tests.xml*.

This file also allows tests to be excluded from a run:

```
<suite name="CDI TCK" verbose="0" configfailurepolicy="continue">
  <test name="CDI TCK">
    ...
    <classes>
      <class name="org.jboss.cdi.tck.tests.context.application.ApplicationContextTest">
        <methods>
          <exclude name="testApplicationScopeActiveDuringServiceMethod"/>
        </methods>
      </class>
    </classes>
    ...
  </test>
</suite>
```

TestNG provides extensive reporting information. Depending on the build tool or IDE you use, the reporting will take a different format. Please consult the TestNG documentation and the tool documentation for more information.

4.6. Configuring your build environment to execute the TCK

It's beyond the scope of this guide to describe in how to set up your build environment to run the TCK. The TestNG documentation provides extensive information on launching TestNG using the Java, Ant, Eclipse or IntelliJ IDEA.

4.7. Configuring your application server to execute the TCK

The TCK makes use of the Java 1.4 keyword `assert`; you must ensure that the JVM used to run the application server is started with assertions enabled. See *Programming With Assertions* [http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html#enable-disable] for more information on how to enable assertions.

Tests within the *jms* test group require some basic Java Message Service configuration. A connection factory, a queue destination for PTP messaging domain and a topic destination for pub/sub messaging domain must be available via JNDI lookup. The corresponding JNDI names are specified with configuration properties - see *Section 4.1, “TCK Properties”*.

Tests within the *persistence* test group require basic data source configuration. The data source has to be valid and JTA-based. The JNDI name of the DataSource is specified with configuration property - see *Section 4.1, “TCK Properties”*.

Tests within the *installedLib* test group require the CDI TCK `cdi-tck-ext-lib` artifact to be installed as a library (see also Java EE 6 specification, section EE.8.2.2 "Installed Libraries").

Tests within the *systemProperties* test group require the following system properties to be set:

Table 4.3.

Name	Value
`cdiTckExcludeDummy`	`true`

Reporting

This chapter covers the two types of reports that can be generated from the TCK, an assertion coverage report and the test execution results. The chapter also justifies why the TCK is good indicator of how accurately an implementation conforms to the JSR 346 specification.

5.1. CDI TCK Coverage Metrics

The CDI TCK coverage has been measured as follows:

- **Assertion Breadth Coverage**

  The CDI TCK provides at least 95% coverage of identified assertions with test cases.

- **Assertion Depth Coverage**

  The assertion depth coverage has not been measured, as, when an assertion requires more than one testcase, these have been enumerated in an assertion group and so are adequately described by the assertion breadth coverage.

- **API Signature Coverage**

  The CDI TCK covers 100% of all API public methods using the Java CTT Sig Test tool.

5.2. CDI TCK Coverage Report

A specification can be distilled into a collection of assertions that define the behavior of the software. This section introduces the CDI TCK coverage report, which documents the relationship between the assertions that have been identified in the JSR 346 specification document and the tests in the TCK test suite.

The structure of this report is controlled by the assertion document, so we'll start there.

5.2.1. CDK TCK Assertions

The CDI TCK developers have analyzed the JSR 346 specification document and identified the assertions that are present in each chapter. Here's an example of one such assertion found in section 2.3.3:

> Any bean may declare multiple qualifier types.

The assertions are listed in the XML file impl/src/main/resources/tck-audit.xml in the CDI TCK distribution. Each assertion is identified by the section identifier of the specification document in which it resides and assigned a unique paragraph identifier to narrow down the location of the assertion further. To continue with the example, the assertion shown above is listed in the tck-audit.xml file using this XML fragment:

```
<section id="declaring_bean_qualifiers" title="Declaring the qualifiers of a bean">
...
<assertion id="d">
  <text>Any bean may declare multiple qualifier types.</type>
</assertion>
...
  </section>
```

The strategy of the CDI TCK is to write a test which validates this assertion when run against an implementation. A test case (a method annotated with `@Test` in a test class) is correlated with an assertion using the `@org.jboss.test.audit.annotations.SpecAssertion` annotation as follows:

```
@Test
@SpecAssertion(section = DECLARING_BEAN_QUALIFIERS, id = "d")
public void testMultipleQualifiers()
{
  Bean<?> model = getBeans(Cod.class, new ChunkyBinding(true), new WhitefishBinding()).iterator().next();
  assert model.getBindings().size() == 3;
}
```

Note

Section identifiers are not used directly. Instead automatically generated constants are applied.

To help evaluate the distribution of coverage for these assertions, the TCK provides a detailed coverage report. This report is also useful to help implementors match tests with the language in the specification that supports the behavior being tested.

5.2.2. Producing the Coverage Report

The coverage report is an HTML report generated as part of the TCK project build. Specifically, it is generated by an annotation processor that attaches to the compilation of the classes in the TCK test suite, another tool from the JBoss Test Utils project. The report is only generated when using Java 6 or above, as it requires the annotation processor.

```
mvn clean install
```

Note

You must run clean first because the annotation processor performs it's work when the test class is being compiled. If compilation is unnecessary, then the assertions referenced in that class will not be discovered.

The report is written to the file target/coverage.html in the same project. The report has five sections:

1. **Chapter Summary** - Lists the chapters (that contain assertions) in the specification document along with total assertions, tests and coverage percentage.
2. **Section Summary** - Lists the sections (that contain assertions) in the specification document along with total assertions, tests and coverage percentage.
3. **Coverage Detail** - Each assertion and the test that covers it, if any.
4. **Unmatched Tests** - A list of tests for which there is no matching assertion (useful during TCK development).
5. **Unversioned Tests** - A list of tests for which there is no `@SpecVersion` annotation on the test class (useful during TCK development).

The coverage report is color coded to indicate the status of an assertion, or group of assertions. The status codes are as follows:

- **Covered** - a test exists for this assertion
- **Not covered** - no test exists for this assertion
- **Problematic** - a test exists but is currently disabled. For example, this may be because the test is under development
- **Untestable** - the assertion has been deemed untestable; a note, explaining why, is normally provided

For reasons provided in the `tck-audit.xml` document and presented in the coverage report, some assertions are not testable.

The coverage report does not give any indication as to whether the tests are passing. That's where the TestNG reports come in.

5.2.3. TestNG Reports

The CDI TCK test suite is really just a TestNG test suite. That means an execution of the CDI TCK test suite produces the same reports as TestNG does. This section will go over those reports and show you where to find each of them.

5.2.3.1. Maven, Surefire and TestNG

When the CDI TCK test suite is executed during the Maven test phase of the TCK runner project, TestNG is invoked indirectly through the Maven Surefire plugin. Surefire is a test execution abstraction layer capable of executing a mix of tests written for JUnit, TestNG, and other supported test frameworks.

Why is this relevant? It means two things. First, it means that you are going to get a summary of the test run on the commandline. Here's the output generated when the tests are run using standalone mode.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running TestSuite
[XmlMethodSelector]
 CLASSNAME:org.jboss.testharness.impl.testng.DisableIntegrationTestsMethodSelector
[XmlMethodSelector] SETTING PRIORITY:0
[XmlMethodSelector]
 CLASSNAME:org.jboss.testharness.impl.testng.ExcludeIncontainerUnderInvestigationMethodSelector
[XmlMethodSelector] SETTING PRIORITY:0
Tests run: 441, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 22.816 sec

Results :

Tests run: 441, Failures: 0, Errors: 0, Skipped: 0
```

Note

The number of tests executed, the execution time, and the output will differ when you run the tests using in-container mode as the CDI TCK requires.

If the Maven reporting plugin that complements Surefire is configured properly, Maven will also generate a generic HTML test result report. That report is written to the file test-report.html in the target/surefire-reports directory of the TCK runner project. It shows how many tests were run, how many failed and the success rate of the test run.

The one drawback of the Maven Surefire report plugin is that it buffers the test failures and puts them in the HTML report rather than outputting them to the commandline. If you are running the test suite to determine if there are any failures, it may be more useful to get this information in the foreground. You can prevent the failures from being redirected to the report using the following commandline switch:

```
mvn test -Dsurefire.useFile=false
```

The information that the Surefire provides is fairly basic and the detail pales in comparison to what the native TestNG reports provide.

5.2.3.2. TestNG HTML Reports

TestNG produces several HTML reports for a given test run. All the reports can be found in the target/surefire-reports directory in the TCK runner project. Below is a list of the three types of reports:

- Test Summary Report
- Test Suite Detail Report
- Emailable Report

The first report, the test summary report, shown below, is written to the file index.html. It produces the same information as the generic Surefire report.

Test results

Suite	Passed	Failed	Skipped	testng.xml
Total	*441*	*1*	*0*	
JSR-299 TCK	441	1	0	Link

The summary report links to the test suite detail report, which has a wealth of information. It shows a complete list of test groups along with the classes in each group, which groups were included and excluded, and any exceptions that were raised, whether from a passed or failed test. A partial view of the test suite detail report is shown below.

JSR-299 TCK

Tests passed/Failed/Skipped:	441/1/0
Started on:	Wed Jul 29 12:53:39 EDT 2009
Total time:	12 seconds (12169 ms)
Included groups:	
Excluded groups:	broken rewrite stub deployment underInvestigation ri-broken

(Hover the method name to see the test class name)

FAILED TESTS		
Test method	**Time (seconds)**	**Exception**
testDecoratorNotResolved	0	java.lang.AssertionError at org.jboss.jsr299.tck.tests.lookup.typesafe.resolution at org.jboss.testharness.AbstractTest.run(AbstractTest.j at org.apache.maven.surefire.testng.TestNGExecutor.run(T at org.apache.maven.surefire.testng.TestNGXmlTestSuite.e at org.apache.maven.surefire.Surefire.run(Surefire.java: at org.apache.maven.surefire.booter.SurefireBooter.runSu at org.apache.maven.surefire.booter.SurefireBooter.main(... Removed 29 stack frames Click to show all stack frames

Test method	
testAbstractApiType	
testAbstractClassDeclaredInJavaNotDiscovered	
testAllBindingTypesSpecifiedForResolutionMustAppearOnBean	
testAmbiguousDependency	

The test suite detail report is very useful, but it borderlines on complex. As an alternative, you can have a look at the emailable report, which is a single HTML document that shows much of the same information as the test suite detail report in a more compact layout. A partial view of the emailable report is shown below.

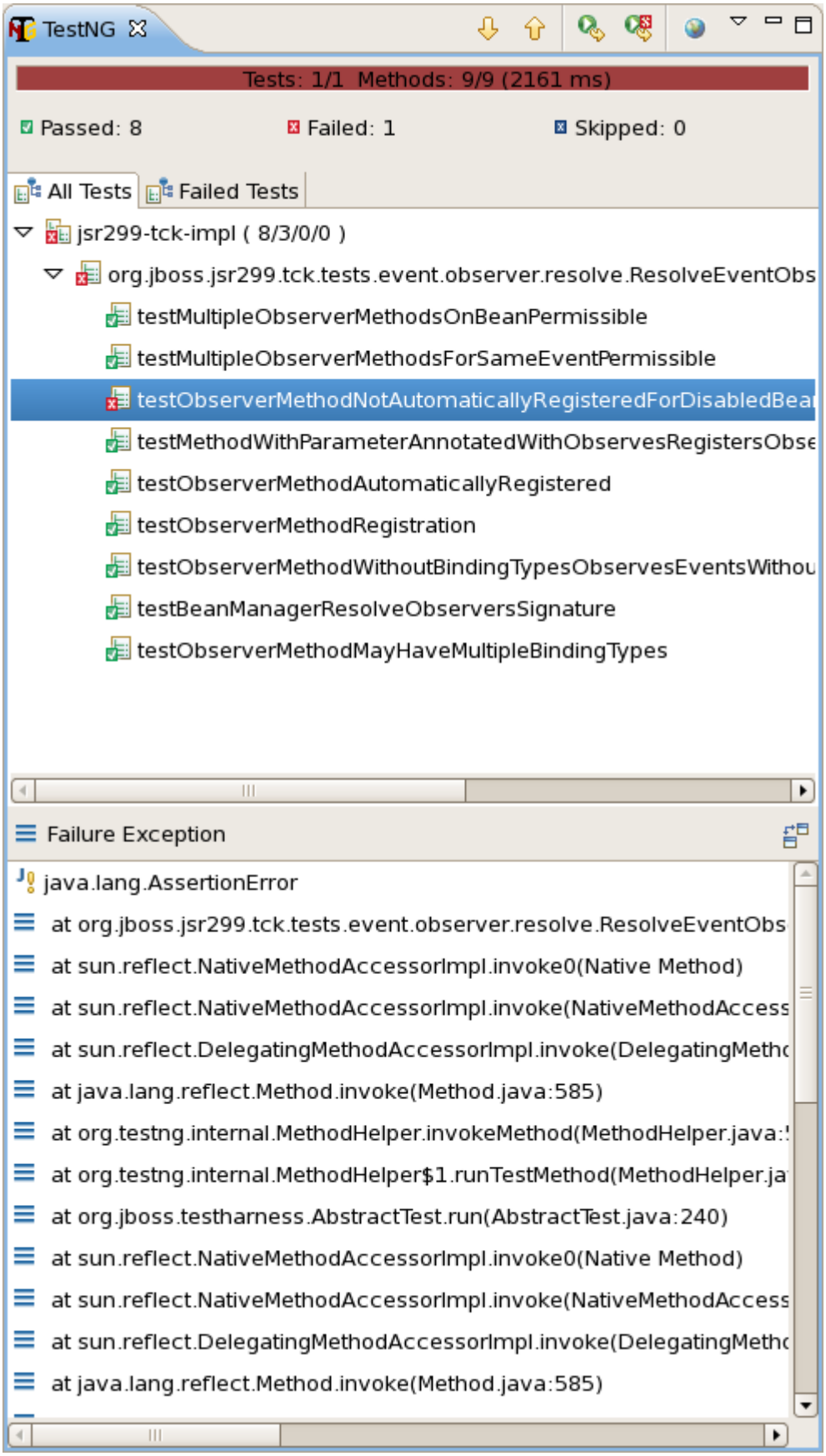
Test	Methods Passed	Scenarios Passed	# skipped	# failed	Total Time	Included Groups	Excluded Groups
JSR-299 TCK	441	441	0	1	12.2 seconds		broken rewrite stub deployment underInvestigation ri-broken

Class
org.jboss.jsr299.tck.tests.lookup.typesafe.resolution.decorator.DecoratorNotResolvedTest
org.jboss.jsr299.tck.tests.context.ContextTest
org.jboss.jsr299.tck.tests.context.DestroyedInstanceReturnedByGetTest
org.jboss.jsr299.tck.tests.context.GetFromContextualTest
org.jboss.jsr299.tck.tests.context.GetOnInactiveContextTest
org.jboss.jsr299.tck.tests.context.GetWithNoCreationalContextTest
org.jboss.jsr299.tck.tests.context.NormalContextTest
org.jboss.jsr299.tck.tests.context.conversation.ConversationBeginTest

Now that you have seen two ways to get test results from the Maven test execution, let's switch over to the IDE, specifically Eclipse, and see how it presents TestNG test results.

5.2.3.3. Test Results in the TestNG Plugin View

After running a test in Eclipse, the test results are displayed in the TestNG plugin view, as shown below.


TestNG
Tests: 1/1 Methods: 9/9 (2161 ms)
Passed: 8
Failed: 1
Skipped: 0
All Tests
Failed Tests
jsr299-tck-impl (8/3/0/0)
org.jboss.jsr299.tck.tests.event.observer.resolve.ResolveEventObs
testMultipleObserverMethodsOnBeanPermissible
testMultipleObserverMethodsForSameEventPermissible
testObserverMethodNotAutomaticallyRegisteredForDisabledBea
testMethodWithParameterAnnotatedWithObservesRegistersObse
testObserverMethodAutomaticallyRegistered
testObserverMethodRegistration
testObserverMethodWithoutBindingTypesObservesEventsWithou
testBeanManagerResolveObserversSignature
testObserverMethodMayHaveMultipleBindingTypes
Failure Exception
java.lang.AssertionError
at org.jboss.jsr299.tck.tests.event.observer.resolve.ResolveEventObs
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMetho
at java.lang.reflect.Method.invoke(Method.java:585)
at org.testng.internal.MethodHelper.invokeMethod(MethodHelper.java:
at org.testng.internal.MethodHelper$1.runTestMethod(MethodHelper.ja
at org.jboss.testharness.AbstractTest.run(AbstractTest.java:240)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMetho
at java.lang.reflect.Method.invoke(Method.java:585)

The view offers two lists. The first is a list of all methods (tests) in the class flagged as either passed or failed. The second is a list of methods (tests) in the class that failed. If there is a test failure, you can click on the method name to get the stacktrace leading up to the failure to display in the lower frame.

You can also find the raw output of the TestNG execution in the IDE console view. In that view, you can click on a test in the stacktrace to open it in the editor pane.

One of the nice features of TestNG is that it can keep track of which tests failed and offer to run only those tests again. You can also rerun the entire class. Buttons are available for both functions at the top of the view.

Part II. Executing and Debugging Tests

In this part you learn how to execute the CDI TCK on the CDI reference implementation (Weld). First, you are walked through the steps necessary to execute the test suite on Weld. Then you discover how to modify the TCK runner to execute the test suite on your own implementation. Finally, you learn how to debug tests from the test suite in Eclipse.

Running the Signature Test

One of the requirements of an implementation passing the TCK is for it to pass the CDI signature test. This section describes how the signature file is generated and how to run it against your implementation.

6.1. Obtaining the sigtest tool

You can obtain the Sigtest tool (at the time of writing the TCK uses version 2.1) from the Sigtest home page at *http://sigtest.java.net/*. The user guide can be found at *http://docs.oracle.com/javame/test-tools/sigtest/2_1/sigtest2.1_usersguide.pdf*.

6.2. Running the signature test

To run the signature test simply change the execution command from `Setup` to `SignatureTest`:

```
java -jar sigtestdev.jar SignatureTest -classpath "%JAVA_HOME%\jre\lib\rt.jar:lib/cdi-api.jar:lib/javax.inject.jar:lib/el-api.jar:lib/jboss-interceptor-api.jar" -Package javax.decorator -Package javax.enterprise -FileName artifacts/cdi-tck-impl-sigtest.sig -static
```

When running the signature test, you may get the following message:
`"The return type java.lang.reflect.Member can't be resolved"`

This can safely be ignored - the important thing is to get the `"STATUS:Passed."` message.

6.3. Forcing a signature test failure

Just for fun (and to confirm that the signature test is working correctly), you can try the following:

1) Edit cdi-api.sig

2) Modify one of the class signatures - in the following example we change one of the constructors for `BusyConversationException` - here's the original:

```
CLSS public javax.enterprise.context.BusyConversationException
cons public BusyConversationException()
cons public BusyConversationException(java.lang.String)
cons public BusyConversationException(java.lang.String,java.lang.Throwable)
cons public BusyConversationException(java.lang.Throwable)
supr javax.enterprise.context.ContextException
hfds serialVersionUID
```

Let's change the default (empty) constructor parameter to one with a `java.lang.Integer` parameter instead:

```
CLSS public javax.enterprise.context.BusyConversationException
cons public BusyConversationException(java.lang.Integer)
cons public BusyConversationException(java.lang.String)
cons public BusyConversationException(java.lang.String,java.lang.Throwable)
cons public BusyConversationException(java.lang.Throwable)
supr javax.enterprise.context.ContextException
hfds serialVersionUID
```

3) Now when we run the signature test using the above command, we should get the following errors:

```
Missing Constructors
--------------------

javax.enterprise.context.BusyConversationException: constructor public
 javax.enterprise.context.BusyConversationException.BusyConversationException(java.lang.Integer)

Added Constructors
------------------

javax.enterprise.context.BusyConversationException: constructor public
 javax.enterprise.context.BusyConversationException.BusyConversationException()


STATUS:Failed.2 errors
```

Executing the Test Suite

This chapter explains how to run the TCK on Weld as well as your own implementation. The CDI TCK uses the Maven Surefire plugin and the Arquillian test platform to execute the test suite. Learning to execute the test suite from Maven is prerequisite knowledge for running the tests in an IDE, such as Eclipse.

7.1. The Test Suite Runner

The test suite is executed by the Maven Surefire plugin during the test phase of the Maven life cycle. The execution happens within a TCK runner project (as opposed to the TCK project itself). Weld includes a TCK runner project that executes the CDI TCK on Weld running inside WildFly 8.x. To execute the CDI TCK on your own CDI implementation, you could modify the TCK runner project included with Weld to use your CDI implementation.

7.2. Running the Tests In Standalone Mode

To execute the TCK test suite against Weld, first switch to the jboss-tck-runner directory in the extracted TCK distribution:

```
cd jsr346/tck/weld/jboss-tck-runner
```

Note

These instructions assume you have extracted the jsr346-related software according to the recommendation given in *The TCK Environment*.

Then execute the Maven life cycle through the test phase:

```
mvn test
```

Without any command-line flags, the test suite is run in standalone mode (activating weld-embedded Maven profile), which means that any test within the *integration* or *javaee-full* TestNG group is excluded. This mode uses the *Weld EE Embedded Arquillian container adapter* to invoke the test within a mock Java EE life cycle and capture the results of the test. However, passing the suite in this mode is not sufficient to pass the TCK as a whole. The suite must be passed while executing using the in-container mode.

7.3. Running the Tests In the Container

To execute the test suite using in-container mode with the JBoss TCK runner, you first have to setup WildFly as described in the *Running the TCK against the CDI RI* callout.

Then, execute the TCK runner with Maven as follows:

```
mvn test -Dincontainer
```

The presence of the `incontainer` property activates an incontainer Maven profile. This time, all the tests in the test suite are executed.

In order to run tests appropriate to the Java EE Web Profile execute:

```
mvn test -Dincontainer=webprofile
```

To specify particular TCK version:

```
mvn test -Dincontainer -Dcdi.tck.version=1.1.0.SP2
```

Note

In order to run the TCK Test Suite in the container an Arquillian container adapter is required. See also *Arquillian reference guide* [https://docs.jboss.org/author/display/ARQ/Containers].

The Arquillian will also start and stop the application server automatically (provided a managed Arquillian container adapter is used).

Since Arquillian in-container tests are executed in a remote JVM, the results of the test must be communicated back to the runner over a container-supported protocol. The TCK utilizes servlet-based protocol (communication over HTTP).

7.4. Dumping the Test Archives

As you have learned, when the test suite is executing using in-container mode, each test class is packaged as a deployable archive and deployed to the container. The test is then executed within the context of the deployed application. This leaves room for errors in packaging. When investigating a test failure, you may find it helpful to inspect the archive after it's generated. The TCK (or Arquillian respectively) can accommodate this type of inspection by "dumping" the generated archive to disk.

The feature just described is activated in the Arquillian configuration file (*Section 4.2, "Arquillian settings"*). In order to export the test archive you'll have to add the `deploymentExportPath` property element inside `engine` element and assign a relative or absolute directory where the test archive should be exported, e.g.:

```
<engine>
   <property name="deploymentExportPath">target/</property>
</engine>
```

Arquillian will export the archive to that location for any test you run.

To enable the export for just a single test, use the VM argument `arquillian.deploymentExportPath`:

```
-Darquillian.deploymentExportPath=target/deployments/
```

Running Tests in Eclipse

This chapter explains how to run individual tests using the Eclipse TestNG plugin. It covers running non-integration tests in standalone mode and integration tests (as well as non-integration tests) in in-container mode. You should be able to use the lessons learned here to debug tests in an alternate IDE as well.

8.1. Leveraging Eclipse's plugin ecosystem

Using an existing test harness (TestNG) allows the tests to be executed and debugged in an Integrated Development Environment (IDE) using available plugins. Using an IDE is also the easiest way to execute a test class in isolation.

The TCK can be executed in any IDE for which there is a TestNG plugin available. Running a test from the CDI TCK test suite using the Eclipse TestNG plugin is almost as simple as running any other TestNG test. You can also use the plugin to debug a test, which is described in the next chapter.

Before running a test from the TCK test suite in Eclipse, you must have the Eclipse *TestNG plugin* [http://testng.org] and either the m2e plugin or an Eclipse project generated using the Maven 2 Eclipse plugin (`maven-eclipse-plugin`). Refer to *Section 3.3, "Eclipse Plugins"* for more information on these plugins.

Note

In order to run the TCK tests in Eclipse you must have CDI TCK and Weld JBoss TCK runner projects imported. Get the source from GitHub repositories *https://github.com/jboss/cdi-tck* and *https://github.com/weld/core*.

With the m2e plugin installed, Eclipse should recognize the CDI TCK projects as valid Eclipse projects (or any Weld project for that matter). Import them into the Eclipse workspace at this time. You should also import the Weld projects if you want to debug into that code, which is covered later.

Tip

If you choose to use the Maven 2 Eclipse plugin (`maven-eclipse-plugin`), you should execute the plugin in both the tck and weld projects:

```
cd tck
mvn clean eclipse:clean eclipse:eclipse -DdownloadSources -
DdownloadJavadocs
cd ../weld
```

```
mvn clean eclipse:clean eclipse:eclipse -DdownloadSources -DdownloadJavadocs
```

8.2. Readying the Eclipse workspace

When setting up your Eclipse workspace, we recommended creating three workings sets:

1. **Weld** - Groups the CDI API and the CDI RI (i.e., Weld) projects
2. **CDI TCK** - Groups the CDI TCK API and the test suite projects
3. **Weld JBoss TCK Runner** - Groups the porting package implementation and TCK runner projects

The dependencies between the projects will either be established automatically by the m2e plugin, based on the dependency information in the pom.xml files, or as generated by the `mvn eclipse:eclipse` command.

Your workspace should appear as follows:

```
Weld
  cdi-api
  weld-core
  ...
CDI TCK
  cdi-tck-api
  cdi-tck-impl
  cdi-tck-parent
Weld JBoss TCK Runner
  weld-jboss-runner-tck11
  weld-porting-package-tck11
```

The tests in the TCK test suite are located in the cdi-tck-impl project. You'll be working within this project in Eclipse when you are developing tests. However, as you learned earlier, there are no references to a CDI implementation in the TCK. So how can you execute an individual test in Eclipse? The secret is that you need to establish a link in Eclipse (not in Maven) between the cdi-tck-impl project and your TCK runner project, which in this case is weld-jboss-runner-tck11 (the project in the jboss-tck-runner/1.1 directory).

Here are the steps to establish the link:

1. Right click on the cdi-tck-impl project
2. Select Build Path > Configure Build Path...

3. Click on the Projects tab
4. Click the Add... button on the right
5. Check the TCK runner project (e.g., weld-jboss-runner-tck11)
6. Click the OK button on the Required Project Selection dialog window
7. Click the OK button on the Java Build Path window

Of course, the weld-jboss-runner-tck11 also depends on the cdi-tck-impl at runtime (so it can actually find the tests to execute). But m2e plugin doesn't distinguish between build-time and runtime dependencies. As a result, we've created a circular dependency between the projects. In all likelihood, Eclipse will struggle (if not fail) to compile one or more projects. How can we break this cycle?

As it turns out, the TCK runner doesn't need to access the tests to build. It only needs its classes, configurations and other dependencies at runtime (when the TestNG plugin executes). Therefore, we can disable *Resolve dependencies from workspace projects* setting on weld-jboss-runner-tck11 project:

1. Right click on the weld-jboss-runner-tck11 project
2. Select Maven
3. Uncheck Resolve dependencies from workspace projects option
4. Click the OK button on the Properties window

As you have learned, the TCK determines how to behave based on the values of system properties or properties defined in META-INF/cdi-tck.properties classpath resources. In order to run the tests, you need to add a properties file to the classpath or define corresponding system properties.

The CDI TCK project conveniently provides the properties file src/test/resources/META-INF/cdi-tck.properties that contains all of the necessary properties for testing in Eclipse. You have to tune the `org.jboss.cdi.tck.libraryDirectory` and `org.jboss.cdi.tck.testDataSource` properties to point to the relative location of the related projects and specify the name of test datasource. The properties should be defined as follows:

- `org.jboss.cdi.tck.libraryDirectory` - the path to the target/dependency/lib directory in the TCK runner project
- `org.jboss.cdi.tck.testDataSource` - the JNDI name of the test datasource, e.g. WildFly 8:

```
org.jboss.cdi.tck.testDataSource=java:jboss/datasources/ExampleDS
```

You are now ready to execute an individual test class (or artifact). Let's start with a test artifact capable of running in standalone mode.

8.3. Running a test in standalone mode

Use *weld-embedded* Maven profile (active by default) in order to run a test in standalone mode.

Tip

If using m2e Eclipse plugin, you can activate/deactivate the profile in Maven section of project propeties.

Note

Note that all TestNG tests that are not included in *integration* and *javaee-full* test groups are considered to be standalone artifacts.

Select a test class containing standalone tests and open it in the Eclipse editor. Now right click in the editor view and select Run As > TestNG Test. The TestNG view should pop out and you should see all the tests in that artifact pass (if all goes well).

Note

If the TCK complains that there is a property missing, close all the projects, open them again, and rebuild. The m2e plugin can be finicky getting everything built correctly the first time.

So far you have executed a test in standalone mode. That's not sufficient to pass the TCK. The test must be executed using in-container mode.

Let's see what has to be done to execute an integration test. This will result in the artifact being deployed to the container, which is WildFly if you are using the JBoss TCK runner.

8.4. Running integration tests

In order to run a test in the container you must explicitly specify following active Maven profiles in JBoss TCK runner Eclipse project properties: `incontainer,!weld-embedded`.

Note

Note that all TestNG tests that are included in *integration* and *javaee-full* test groups are considered to be integration tests and must be run in in-container mode.

> *javaee-full* TestNG test group contains tests that require full Java EE platform (EAR packaging, JAX-WS, EJB timers, etc.).

Select an integration test (a class that extends `org.jboss.cdi.tck.AbstractTest` and open it in your Eclipse editor. Right click in the editor view and select Run As > TestNG Test.

You have now mastered running the CDI TCK against Weld using both Maven and within Eclipse. Now you're likely interested in how to debug a test so that you can efficiently investigate test failures.

Debugging Tests in Eclipse

This chapter explains how to debug standalone and integration tests from the TCK test suite in Eclipse. You should be able to use the lessons learned here to debug tests in an alternate IDE as well.

9.1. Debugging a standalone test

There is almost no difference in how you debug a standalone test from how you run it. With the test class open in the Eclipse editor, simply right click in the editor view and select Debug As > TestNG Test. Eclipse will stop at any breakpoints you set just like it would with any other local debug process.

If you plan to step into a class in the Weld implementation (or any other dependent library), you must ensure that the source is properly associated with the library. Below are the steps to follow to associate the source of Weld with the TestNG debug configuration:

1. Select the Run > Debug Configurations... menu from the main menubar
2. Select the name of the test class in the TestNG category
3. Select the Source tab
4. Click the Add... button on the right
5. Select Java Project
6. Check the project that contains the class you want to debug (e.g., weld-core)
7. Click OK on the Project Selection window
8. Click Close on the Debug Configurations window

You'll have to complete those steps for any test class you are debugging, though you only have to do it once (the debug configuration hangs around indefinitely).

Again, running a test in standalone mode isn't enough to pass the TCK and cannot be used to run or debug an integration test. Let's look at how to debug a test running in the context of the container.

9.2. Debugging an integration test

In order to debug an integration test, or any test run using in-container mode, the test must be configured to run in-container, as described in *Section 8.4, "Running integration tests"*, and you must attach the IDE debugger to the container. That puts the debugger on both sides of the fence, so to speak.

Since setting up a test to run in-container has already been covered, we'll look at how to attach the IDE debugger to the container, and then move on launching the test in debug mode.

9.2.1. Attaching the IDE debugger to the container

There are two ways to attach the IDE debugger to the container. You can either start the container in debug mode from within the IDE, or you can attach the debugger over a socket connection to a standalone container running with JPDA enabled.

The Eclipse Server Tools, a subproject of the Eclipse Web Tools Project (WTP), has support for launching most major application servers, including WildFly 8. However, if you are using WildFly, you should consider using JBoss Tools instead, which offers tighter integration with JBoss technologies. See either the *Server Tools documentation* [http://www.eclipse.org/webtools/server/server.php] or the *JBoss Tools documentation* [http://docs.jboss.org/tools/] for how to setup a container and start it in debug mode.

See *this blog entry* [http://justinjohnson.org/development/configuring-remote-debugging-in-jboss-as-7-and-eclipse/] to learn how to start WildFly with JPDA enabled and how to get the Eclipse debugger to connect to the remote process.

9.2.2. Launching the test in the debugger

Once Eclipse is debugging the container, you can set a breakpoint in the test and debug it just like a standalone test. Let's give it a try.

Open a test in the Eclipse editor, right click in the editor view, and select Debug As > TestNG Test (this works for the container started in debug mode from within the IDE) or run the TestNG test and debug Remote Java Application (remote debug configuration) in the same time (when attaching the debugger over a socket connection to a container). This time when the IDE hits the breakpoint, it halts the JVM thread of the container rather than the thread that launched the test.

Remember that if you need to debug into dependent libraries, the source code for those libraries will need to be registered with the TestNG debug configuration as described in the first section in this chapter.