# JTiger 2.1 User Documentation

Build Number: 0376
Build Time: 2006-07-28 01:50.16.218 CET (GMT +1)
http://jtiger.org/

## Contents

# 1. Introduction

### 1.1 This Document

This document is intended as an introduction to the JTiger unit test framework and tools. This document is complemented by the JTiger Javadoc API Documentation, which is included with the JTiger package download. It is intended that the Javadoc is used as a reference during unit test development with JTiger. This document will inevitably contain some redundancy with the Javadoc reference material.

This document is available in several file formats:

- FO (XML Formatting Objects)
- HTML (HyperText Markup Language)
- MIF (Media Interchange Format)
- PDF (Portable Document Format)
- PS (PostScript)
- TXT (Plain Text)

### 1.2 What is JTiger?

JTiger is a unit test framework and tools for the Java 2 Platform. The framework provides a useful abstraction on which to write unit test fixtures and unit test cases. JTiger tools provide functionality that is often desired in software unit testing environments. JTiger development encourages Test Driven Development, though it doesn't mandate it, and any unit testing software development technique is sufficient. JTiger makes heavy use of the Java Programming Language 1.5 features; annotations, generics, variable argument lists, and more. JTiger encourages developers to document unit test fixtures and unit test cases in order to provide a robust, and easily maintained unit test and regression harness.

The JTiger framework provides a published and documented API should the need arise to extend its functionality. An example of extending the JTiger framework, is the ability to execute test cases that are written using the JUnit unit test framework. The JUnit plugin implementation classes are included as part of the JTiger framework.

### 1.3 Why use JTiger?

JTiger makes every effort to ensure a robust unit test harness by providing a rich set of tools, and a reliable, usable framework on which to develop unit test cases. JTiger is an implementation that is based on improvements in software development methodologies, such as eXtreme Programming (XP), which have matured since their inception.

JTiger has been implemented using the same techniques that it encourages, specifically, Test Driven Development. JTiger includes its own unit test and regression harness with 100% method coverage. This aids in new or modified requirements for JTiger that arise in the unforeseen future. The JTiger source code base has been designed to ensure the absolute maximum possible amount of decoupling of components and encapsulation such that future enhancements will not have a negative impact on future versions on JTiger.

#### 1.4 System Requirements

JTiger requires that tests are executed using a Java Virtual Machine version 1.5 or higher. This does not mean that the code under test must be written using Java 1.5 language features. The software under test may target any JVM version such as 1.2, 1.3 or 1.4. It is merely the test run that must execute under a JVM version 1.5. JTiger is capable of executing test cases that have been written using the JUnit test framework.

#### 1.5 Download Package Structure

- `/jtiger.jar`

  The JTiger Java archive binary file. This file contains the classes that are necessary for using JTiger. Make this file available to your Java Runtime Environment to use JTiger. This typically involves specifying jtiger.jar in the Java classpath.

- `/src/`

  The JTiger Java source code directory. This directory contains the source code to JTiger.

- `/test-src/`

  The JTiger Java test source directory. This directory contains the source

code of the unit test harness for JTiger. This source code is useful for determining how a particular feature of JTiger is intended to be used or how it works and is considered a formal and complete specification of the requirements of JTiger.

- `/doc/javadoc/`

  The JTiger Javadoc API documentation directory. This directory contains the Java API documentation to every public JTiger API feature. The JTiger Java API documentaton is intended to be used as a reference when using JTiger. Of particular interest for reference is the package documentation for each JTiger package which gives an overview of the objective of that particular package.

- `/doc/userdoc/`

  The JTiger user documentation directory. This directory contains this document in a number of different file formats.

- `/doc/test-report/`

  The JTiger test result report directory. This directory contains a JTiger test report using the HTML fixture results handler. A 100% success rate in the report indicates that JTiger functions correctly and as specified. You may generate your own test report in the same format by using the JTiger package org.jtiger.report.html. See the package documentation for more details.

- `/doc/test-coverage-report/`

  The JTiger test code coverage report directory. This directory contains a report indicating how much of the JTiger source code was executed during unit test execution. The JTiger test source code aims to achieve 100% class coverage and 100% method coverage during unit test execution i.e. all JTiger methods or all JTiger classes are executed at least once during unit test execution.

- `/samples/`

  The JTiger sample source code directory. This directory contains all sample source code that is used in JTiger documentation.

# 2. Writing Test Fixtures

### 2.1 What is a Test Fixture?

A JTiger test fixture is a logical grouping of test cases. A JTiger test fixture is a single Java class that contains zero or more test cases, which are methods in that class. A simple test fixture with one test case follows:

```
package org.jtiger.samples;

import org.jtiger.framework.Test;
import static org.jtiger.assertion.Basic.assertEqual;

public final class ExampleTestFixture
{
    @Test
    public void exampleTestCase()
    {
        // assert that two plus two equals four.
        assertEqual(4, 2 + 2);
    }
}
```

A JTiger test fixture class may optionally be annotated with org.jtiger.framework.Fixture. This annotation is used only for documentation purposes - it is not a requirement that test fixtures use it. A test fixture class may specify the categories that its test cases are in by using the org.jtiger.framework.Category annotation. A test case method may be in zero or more categories. A JTiger test fixture class must have a constructor that takes no arguments or a constructor that takes a single String argument.

### 2.2 What is a Test Case?

A JTiger test case method is potentially any method that is present in a test fixture class. A JTiger test case method must take no arguments. If a method is specified as a test case method and it takes one or more arguments, execution of that test case will have a 'Ignored (Cannot Invoke)' test case result. You can specify a method as a test case method by using the default TestDefinition implementation, where the method is annotated with the org.jtiger.framework.Test annotation. A test case method that has been written using the JUnit test framework can be executed by using the JUnit TestDefinition implementation, which specifies a test case method as any method that has a name beginning with "test". A test case method may have a return value. If the test case method has a return value that is not null and the method doesn't throw

an exception during test case execution, the return value is used as the test case result message. If an exception is thrown during test case execution, the exception message is used as the test case result message.

The execution of a JTiger test case method can result in one of six states:

1. `Success`

   The test case method executed successfully without an exception, or, if an exception was expected, the exception occurred. A test case method can specify that it is expecting an exception to occur by using the org.jtiger.framework.ExpectException annotation.

2. `Ignored (Annotated)`

   The test case method was not executed because it is annotated with org.jtiger.framework.Ignore.

3. `Ignored (Cannot Invoke)`

   The test case method was not executed because it does not meet the requirements for a test case method. Specifically, it takes one or more arguments. It is a requirement that a JTiger test case method takes zero arguments.

4. `Failure (Set Up)`

   The test case method was not executed because it could not be set up correctly. A set up method in the test fixture threw an exception when it was executed.

5. `Failure`

   The test case method was executed and resulted in failure. A test case method results in failure if an exception occurs during execution that it wasn't expecting. A test case method can specify that it is expecting an exception to occur by using the org.jtiger.framework.ExpectException annotation. If the expected exception occurs, the test case result is 'Success'.

6. `Failure (Tear Down)`

The test case method was executed, however, it could not be torn down correctly. A tear down method in the test fixture threw an exception when it was executed.

A JTiger test case result is resolved using these rules, in the following order:

1. If the test case method is annotated with org.jtiger.framework.Ignore, the test case result is 'Ignored (Annotated)'.
2. If the test case method takes one or more arguments, the test case result is 'Ignored (Cannot Invoke)'.
3. If an exception occurs during execution of any of the set up methods in the test fixture, the test case result is 'Failure (Set Up)'.
4. If an exception occurs during execution of any of the tear down methods in the test fixture, which occurs after test case method execution, the test case result is 'Failure (Tear Down)'.
5. If an exception occurs during execution of the test case method and that exception type was not expected, the test case result is 'Failure'.
6. If an exception occurs during execution of the test case method and that exception type was expected, the test case result is 'Success'.
7. If an exception does not occur during execution of the test case method and an exception was expected, the test case result is 'Failure'.
8. If an exception does not occur during execution of the test case method and an exception was not expected, the test case result is 'Success'.

Following is a test fixture that contains five (5) test cases. After execution of these test cases, each has a result of 'Success'. Note that the assertion methods (those whose name begins with "assert") may throw a org.jtiger.assertion.AssertionException, however, they never do because all of the assertions that are made, hold true.

```
package org.jtiger.samples;

import org.jtiger.framework.Test;
import org.jtiger.framework.ExpectException;
import org.jtiger.assertion.ObjectFactory;
import static org.jtiger.assertion.Basic.assertEqual;
import static org.jtiger.assertion.Serialization.assertSerializes;
import static org.jtiger.assertion.Serialization.assertSerializesEqual;
import static
org.jtiger.assertion.ObjectFactoryContract.assertObjectFactoryFillsContract;
import static
org.jtiger.assertion.EqualsMethodContract.assertEqualsMethodFillsContract;
import static
org.jtiger.assertion.HashCodeMethodContract.assertHashCodeMethodFillsContract;

public final class SomeTestCases
{
    public SomeTestCases()
    {

    }

    @Test
    public void simpleTestCase1()
    {
        // Assert that two plus two equals four.
        assertEqual(4, 2 + 2);
    }

    @Test
    public void simpleTestCase2()
    {
        // Assert that an instance of the Integer
        // class serializes and deserializes.
        assertSerializes(new Integer(7));
    }

    @Test
    public void simpleTestCase3()
    {
        // Assert that an instance of the Integer
        // class serializes and deserializes to be equal.
        assertSerializesEqual(new Integer(7));
    }

    @Test
    public void simpleTestCase4()
    {
        // Assert that the Integer class meets the Object equals
        // and hashCode method contracts.
        // This test is only as good as the test data returned by
        // the ObjectFactory methods.
        final ObjectFactory<Integer> factory = new
ObjectFactory<Integer>()
        {
            public Integer newInstanceX()
            {
                return new Integer(7);
            }

            public Integer newInstanceY()
            {
                return new Integer(8);
            }
```

```
        };

        // Assert that the test data is valid and meets
        // the ObjectFactory contract.
        assertObjectFactoryFillsContract(factory);
        // Assert that the Object equals method contract is met.
        assertEqualsMethodFillsContract(factory);
        // Assert that the Object hashCode method contract is met.
        assertHashCodeMethodFillsContract(factory);
    }

    @Test
    @ExpectException(NullPointerException.class)
    public void simpleTestCase5()
    {
        // Expect a NullPointerException to occur.
        new String((StringBuffer)null);
    }
}
```

### 2.3 Best Practices

- Use the org.jtiger.framework.Fixture annotation on test fixtures. Although it is not required, this annotation provides documentation that is used in test reports, and for readers of the test source code.
- Use the appropriate assertions. For example, when testing for equality, use org.jtiger.assertion.Basic.assertEqual.

```
package org.jtiger.samples;

import org.jtiger.framework.Test;
import static org.jtiger.assertion.Basic.assertEqual;
import static org.jtiger.assertion.Basic.assertTrue;

public final class PreferredAssertionTestFixture
{
    @Test
    public void preferredTestCase()
    {
        // assert that two plus two equals four.
        // preferred.
        assertEqual(4, 2 + 2);
    }

    @Test
    public void notPreferredTestCase()
    {
        // assert that two plus two equals four.
        // not preferred.
        assertTrue(2 + 2 == 4);
    }
}
```

The appropriate assertion for the task is a more formal specification of the software requirements. It is also less prone to error.

- Categorize test cases using the org.jtiger.framework.Category annotation. This annotation can be placed on test fixture classes or test case methods. If test cases belong to a category, a subset of them can be executed by passing the appropriate regular expression at test execution time.

- Use the org.jtiger.framework.Ignore annotation during test case development instead of commenting out test cases. This means that you will be provided with a test result of 'Ignored (Annotated)' instead of silently ignoring what should have been a test case, in the circumstance that you forget to uncomment it.

- Use a static import for the methods in the assertions package. This results in test cases that are clear and easy to read.

- Some assertions require an instance of ObjectFactory (org.jtiger.assertion.ObjectFactory) as a parameter. The ObjectFactory interface demands that implementations meets certain aspects of a contract, which are specified in the ObjectFactory Javadoc API documentation. Before using an instance of ObjectFactory for making an assertion, assert that your ObjectFactory implementation meets the contract. This can be achieved with a call to the org.jtiger.assertion.ObjectFactoryContract.assertObjectFactoryFillsContract method. This means that any assertions that are made on the ObjectFactory instance can be considered genuine, as opposed to being incorrect because of faulty test data that is returned by the ObjectFactory implementation.

# 3. Executing Test Fixtures

### 3.1 Executing Test Fixtures Using the API

Test fixtures can be executed using the JTiger API. The org.jtiger.framework.FixturesRunnerFactory class is considered the core of the framework returning instances of org.jtiger.framework.FixturesRunner. Using one of these instances, you can run a given configuration (org.jtiger.framework.FixturesRunnerConfig) and have a test result (org.jtiger.framework.FixtureResults) returned.

Following is a class that contains a main method and is also a test fixture that is executed by the fixtures runner. It is strongly recommended that the JTiger Javadoc be used as a reference when developing with the JTiger API.

```
package org.jtiger.samples;

import static
org.jtiger.framework.FixturesRunnerConfigFactory.newFixturesRunnerConfig;
import static
org.jtiger.framework.FixturesRunnerFactory.newFixturesRunner;
import static org.jtiger.framework.SequenceFactory.newSequence;
import static org.jtiger.assertion.Basic.assertTrue;
import static org.jtiger.assertion.Basic.assertFalse;
import org.jtiger.framework.FixturesRunnerConfig;
import org.jtiger.framework.Test;
import org.jtiger.framework.Category;
import org.jtiger.framework.FixtureResults;
import org.jtiger.framework.FixturesRunner;
import org.jtiger.framework.RunnerException;
import org.jtiger.framework.FixtureResult;
import org.jtiger.framework.TestResult;

public final class ExampleAPIUsage
{
    public ExampleAPIUsage()
    {

    }

    // The class that uses the JTiger API is the same
    // class that is used as a test fixture.
    // This is not ideal - typically, the test run would
    // include classes from some other source.
    @Test
    @Category("samples")
    public String testCase()
    {
        assertTrue(true);
        assertFalse(false);
        return "true is true and false is false";
    }

    public static void main(String... args) throws RunnerException
    {
        Class<?>[] fixtureClasses = new
Class<?>[]{ExampleAPIUsage.class};

        FixturesRunnerConfig config =
newFixturesRunnerConfig(newSequence(fixtureClasses),
            null, null, false, "samples");
        FixturesRunner runner = newFixturesRunner();
        FixtureResults results = runner.run(config);

        for(FixtureResult result : results)
        {
            for(TestResult tr : result)
            {
                System.out.println(tr.getTestResultType() + " : " +
tr.getMessage());
            }
        }
    }
```

```
}
```

#### 3.2 Executing Test Fixtures From the Command Line

Test fixtures can be executed from the command line by passing the configuration information using command line arguments. The JTiger Javadoc for the org.jtiger.framework.FixturesRunnerMain class provides valuable information for execution from the command line. The JTiger JAR file can be executed using the -jar switch, however, remember that any classpath setting (CLASSPATH environment variable or -classpath command line switch) is ignored by the JVM system class loader when this switch is used. Two example commands follow:

```
java -jar jtiger.jar -fixtureClasses org.jtiger.ant.TestCategory
org.jtiger.samples.ExampleAPIUsage -categories samples
```

```
java -classpath jtiger.jar org.jtiger.framework.FixturesRunnerMain
-fixtureClasses org.jtiger.ant.TestCategory
    org.jtiger.samples.ExampleAPIUsage -categories samples -result ~html
-resultParams reports
```

If a command line argument is malformed or no command line arguments are given, usage information will be sent to the standard error stream. All command line arguments are optional. It is recommended that the -fixtureClasses command line argument always be used. The available command line arguments correspond to the attributes of an instance of org.jtiger.framework.FixturesRunnerConfig with the addition of an implementation of org.jtiger.framework.FixtureResultsHandler to handle the results of the test fixture execution. The available command line arguments and their meanings follow:

- `-fixtureClasses`

  This command line argument is followed by one or more test fixture class names whose test case methods are executed during the test run.

- `-definitionClass`

  This command line argument is followed by the class name to use that defines a test case method. This class must implement the org.jtiger.framework.TestDefinition interface, otherwise, the default Test

Definition implementation is used. By default, a test case method is defined as one that is annotated with org.jtiger.framework.Test.

- `-sutdClass`

  This command line argument is followed by the class name to use to perform test case set up and tear down. This class must implement the org.jtiger.framework.SetUpTearDown interface, otherwise, the default Set Up Tear Down implementation is used. By default, test case set up occurs by executing all methods that are annotated with org.jtiger.framework.SetUp and test case tear down occurs by executing all methods that are annotated with org.jtiger.framework.TearDown.

- `-haltOnFailure`

  This command line argument is used to specify whether test execution should halt if a test failure is encountered. The default value is "false".

- `-junit`

  This command line argument is used as a shortcut for specifying that the included test fixtures have been written using the JUnit test framework. If this command line argument is specified, the values of the -definitionClass and -sutdClass command line arguments are ignored, since they are implicitly set to values representing the implementation classes that are used for JUnit test case execution.

- `-categories`

  This command line argument is followed by one or more regular expressions of categories of test cases that are to be executed. A test case is put into one or more categories by annotating the test case method or the containing test fixture with org.jtiger.framework.Category. By default, all test cases are executed if no categories are specified, including those that do not explicitly specify a category.

- `-result`

  This command line argument is followed by the class name that handles the test fixture results. There are four special, case-insensitive values that may be used; "~html", "~xml", "~text" or "~failure" for the name attribute to use the Fixture Results Handler implementations that are included with JTiger. This class must implement the org.jtiger.framework.FixtureResultsHandler

interface or be one of the four special values, otherwise, the default Fixture Results Handler implementation is used. By default, fixture results are summarized and sent to the standard output stream.

- `-resultParams`

  This command line argument is followed by one or more parameters that are passed to the fixture results handler implementation. If the -result command line argument is not used, this command line argument is ignored, since the default Fixture Results Handler implementation does not use any parameters that are passed. Some JTiger Fixture Results Handler implementations output a HTML, XML, or plain text report to a file or directory. The file or directory name is passed as a parameter to the JTiger Fixture Results Handler implementation.

#### 3.3 Executing Test Fixtures From the Apache Ant Task

Test fixtures can be executed from the JTiger Apache Ant task, which is represented by the org.jtiger.ant.JTigerTask class. The Javadoc for this class provides valuable information regarding the attributes and elements of the task. The attributes and elements of this task correspond to the attributes of an instance of org.jtiger.framework.FixturesRunnerConfig with the addition of an implementation of org.jtiger.framework.FixtureResultsHandler to handle the results of the test fixture execution and an element for providing the environment of the Java Virtual Machine that is used.

The following example is an excerpt from the Apache Ant build file that is used to execute the JTiger Self Test Fixtures:

```
<taskdef name="jtiger" classname="org.jtiger.ant.JTigerTask"
classpathref="project.class.path"/>
<jtiger haltonfailure="false">
    <category regex="org\.jtiger\..*"/>
    <fixtures>
        <fileset dir="test-src">
            <include name="**/**.java"/>
        </fileset>
    </fixtures>
    <result name="~html">
        <param value="report/test-report"/>
    </result>
    <java failonerror="true">
        <classpath refid="project.class.path"/>
    </java>
</jtiger>
```

### 3.4 The Set Up Tear Down implementations

Set Up Tear Down implementations are represented by the org.jtiger.framework.SetUpTearDown interface. JTiger includes two implementations of this interface; the default implementation and the implementation that is used for executing test cases that have been written using the JUnit test framework. It is possible to write a custom implementation to use for test execution that is invoked at the time of test case set up and test case tear down.

The Set Up Tear Down implementation class is returned from the getSutdClass method of the org.jtiger.framework.FixturesRunnerConfig class, or null is returned to use the default Set Up Tear Down implementation. A custom implementation class must have a constructor that takes no arguments or a constructor that takes a single String argument.

The default Set Up Tear Down implementation class is represented by the org.jtiger.framework.DefaultSetUpTearDown class. This class performs test case set up by invoking any methods that are annotated with org.jtiger.framework.SetUp and performs test case tear down by invoking any methods that are annotated with org.jtiger.framework.TearDown. If there are two or more methods that are annotated with @SetUp, or two or more methods that are annotated with @TearDown, it is not guaranteed which order the methods will be executed during test case set up or test case tear down.

The JUnit Set Up Tear Down implementation class is represented by the org.jtiger.framework.junit.JUnitSetUpTearDown class. This class performs test case set up by invoking the first method in the superclass hierarchy that is called "setUp" and takes no arguments, and performs test case tear down by invoking the first method in the superclass hierarchy that is called "tearDown" and takes no arguments.

### 3.5 The Test Definition implementations

Test Definition implementations are represented by the org.jtiger.framework.TestDefinition interface. JTiger includes two implementations of this interface; the default implementation and the implementation that is used for executing test cases that have been written using the JUnit test framework. It is possible to write a custom implementation to use for test execution that is invoked to determine how a method is defined as a test case.

The Test Definition implementation class is returned from the getDefinitionClass method of the org.jtiger.framework.FixturesRunnerConfig class, or null is returned to use the default Test Definition implementation. A custom implementation class must have a constructor that takes no arguments or a constructor that takes a single String argument.

The default Test Definition implementation class is represented by the org.jtiger.framework.DefaultTestDefinition class. This class defines a test case as any method that is annotated with org.jtiger.framework.Test.

The JUnit Test Definition implementation class is represented by the org.jtiger.framework.junit.JUnitTestDefinition class. This class defines a test case as any method that has a name beginning with "test".

#### 3.6 The Fixture Results Handler implementations

Fixture Results Handler implementations are represented by the org.jtiger.framework.FixtureResultsHandler interface. JTiger includes four implementations of this interface; the default implementation, the HTML report implementation, the XML report implementation and the plain text report implementation.

The default Fixture Results Handler implementation class is represented by the org.jtiger.framework.DefaultFixtureResultsHandler class. This class handles test case results by sending a summary of the results to the standard output stream. For example, "Success: [672] Failure: [0] Ignored: [0]". A custom implementation class must have a constructor that takes no arguments or a constructor that takes a single String argument.

The HTML report Fixture Results Handler implementation class is represented by the org.jtiger.report.html.HtmlFixtureResultsHandler class. This class handles test case results by creating a HTML report in the directory that is passed as a parameter to the Fixture Results Handler. If no parameter is passed the current directory is used. The HTML files contain detailed test case result information in HTML format. The report index is a file called 'index.html'.

The XML report Fixture Results Handler implementation class is represented by the org.jtiger.report.xml.XmlFixtureResultsHandler class. This class handles test case results by producing an XML report file. The name of the file is passed as a parameter to the Fixture Results Handler. If no parameter is passed a file called 'result.xml' in the current directory is used. The XML file contains detailed test

case result information in XML format.

The plain text report Fixture Results Handler implementation class is represented by the org.jtiger.report.text.TextFixtureResultsHandler class. This class handles test case results by writing a plain text file. The name of the file is passed as a parameter to the Fixture Results Handler. If no parameter is passed a file called 'result.txt' in the current directory is used. The text file contains detailed test case result information in plain text format.

#### 3.7 Halting Test Execution When a Failure Occurs

If a failure occurs during test execution, any pending test cases can be halted from execution and a result returned immediately. By default, test execution is not halted when a failure is encountered. A failure is defined as any one of the failure test case results; 'Failure', 'Failure (Set Up)', or 'Failure (Tear Down)'.

The halt on failure boolean value is returned from the isHaltOnFailure method of the org.jtiger.framework.FixturesRunnerConfig class.

#### 3.8 Test Case Categories

Test cases belong to a category by annotating the test case method with org.junit.framework.Category or the test fixture class that contains it. A test case method is said to be in all of the categories that form the union where the @Category annotation appears on the test case method, and its test fixture.

```
package org.jtiger.samples;

import org.jtiger.framework.Category;
import org.jtiger.framework.Test;

@Category({"a", "b"})
public final class ExampleCategory
{
    @Test
    @Category("c")
    public void m1()
    {

    }

    @Test
    public void m2()
    {

    }
```

```
    @Test
    @Category({"c", "d"})
    public void m3()
    {

    }
}
```

In the above example, the method 'm1' is in the categories 'a', 'b' and 'c'. The method 'm2' is in the categories 'a' and 'b'. The method 'm3' is in the categories 'a', 'b', 'c' and 'd'.

Categories are specified at test execution time as regular expressions. If no categories are specified, all test case methods that are in any category will be executed. Test case methods that are in no category can be executed only by not specifying any category regular expressions. It is important to remember that regular expressions have characters that are not used as their literal meaning.

In the above example, given the regular expression, "[cdef]" as a category for execution, the methods 'm1' and 'm2' will be included because they are the only two methods that are in a category that matches the regular expression.

## 4. Making Assertions in Test Cases

JTiger includes a rich API for making assertions in your test cases. Assertions range from the very basic, "assert true", to the complex, "assert the Object equals method contract". The JTiger assertions API is represented by the org.jtiger.assertion package. It is strongly recommended that you consult the JTiger Javadoc API documentation when using the JTiger assertions.

Assertions typically require some argument on which to make the assertion. For example, the "assert true" (org.jtiger.assertion.Basic.assertTrue) assertion takes a single boolean argument. If this argument is not true, the test case fails by throwing a org.jtiger.assertion.AssertionException. It is not recommended to use an AssertionException in an @ExpectException annotation or to use a java.lang.RuntimeException, java.lang.Exception or java.lang.Throwable with subclasses set to the value true if it possible that your test case may throw an AssertionException i.e. you use any of the JTiger assertions.

All JTiger assertions take a variable argument list of type java.lang.Object. Any arguments passed have their toString method called and appended in sequence to form a single test case message. This message is used in test reports and is

available from the API when a test result is returned and in JTiger test reports.

```
assertTrue(true);
assertEqual(new Integer(7), new Integer(7), "the Integer 7 is not equal
to the Integer 7");
assertNull(null, "null", " is not null");
assertSerializes(new Float(7.6F), new StringBuilder()
    .append("The Float with the value")
    .append(7.6F).append("did not serialize"));
```

## 5. The Apache Ant Task

JTiger includes an Apache Ant task for starting a test execution from an Apache Ant build file. The JTiger Apache Ant task is represented by the org.jtiger.ant.JTigerTask class. The task allows the following elements and attributes:

- `<fixtures> element`

  This element may contain zero or more a fixture subelements or zero or more fileset subelements. This element is used to specify the names of the test fixtures to execute. Files that are matched by fileset subelements are converted to a class name by removing the .java or .class file extension and changing the directory separators to dots. If a file name does not have the .java or .class suffix, it is ignored. For example, the file name "org/jtiger/MyClass.class" would be converted to the class name "org.jtiger.MyClass". The fixture subelement accepts a single attribute called classname.

- `definitionClass attribute`

  This attribute is used to specify the class name to use as the Test Definition implementation during test execution. If this attribute is not present, the default Test Definition implementation is used.

- `sutdClass attribute`

  This attribute is used to specify the class name to use as the Set Up Tear Down implementation during test execution. If this attribute is not present, the default Set Up Tear Down implementation is used.

- `<category> element`

This element may appear zero or more times to specify the test category regular expressions to use during test execution. If no category elements are present, all test cases are executed. This element accepts a single attribute called regex.

- `haltOnFailure attribute`

  This attribute is used to specify whether test execution should halt if a test failure is encountered. The default value is "false". This attribute can be set to true by giving it a value of "true", "yes" or "on".

- `JUnit attribute`

  This attribute is used as a shortcut for specifying that the included test fixtures have been written using the JUnit test framework. If this attribute has a value of "true", "yes" or "on", the values of the definitionClass and sutdClass attributes are ignored, since they are implicitly set to values representing the implementation classes that are used for JUnit test case execution.

- `<result> element`

  This element specifies the class name of the Fixture Results Handler to use for test execution. There are four special, case-insensitive values that may be used; "~html", "~xml", "~text" or "~failure" for the name attribute to use the Fixture Results Handler implementations that are included with JTiger. These values are case-insensitive, and each of them requires at least one parameter, which is the name of the file or directory to write the test results report to. A Fixture Results Handler parameter is passed using the param subelement. The param subelement accept a single attribute called value.

- `<java> element`

  This element specifies the Java Virtual Machine environment that is used for test execution. The Java Virtual Machine is a forked process from the Java Virtual Machine that the Ant task is running in to provide an independant execution environment. This element provides most, but not all, of the same subelements and attributes as the core Ant <java> task.

Following are some example excerpts from an Apache Ant build file.

- ```
  <taskdef name="jtiger" classname="org.jtiger.ant.JTigerTask"
  classpathref="project.class.path"/>
  <jtiger junit="true">
      <fixtures>
          <fileset dir="test-src">
              <include name="**/**.java"/>
          </fileset>
      </fixtures>
      <java failonerror="true">
          <classpath refid="project.class.path"/>
      </java>
  </jtiger>
  ```

- ```
  <taskdef name="jtiger" classname="org.jtiger.ant.JTigerTask"
  classpathref="project.class.path"/>
  <jtiger haltonfailure="true" sutdClass="com.foo.MySetUpTearDown"
  definitionClass="com.foo.MyTestDefinition">
      <category regex="Database test cases"/>
      <fixtures>
          <fixture classname="com.foo.MyTestFixture"/>
          <fixture classname="com.foo.MyOtherTestFixture"/>
          <fileset dir="test-src">
              <include name="**/**.java"/>
          </fileset>
      </fixtures>
      <result name="com.foo.MyFixtureResultsHandler">
          <param value="a value"/>
      </result>
      <java failonerror="true">
          <classpath refid="project.class.path"/>
      </java>
  </jtiger>
  ```

- ```
  <taskdef name="jtiger" classname="org.jtiger.ant.JTigerTask"
  classpathref="project.class.path"/>
  <jtiger haltonfailure="true">
      <category regex="[Aa].*"/>
      <category regex="[Bb].*"/>
      <category regex="[Cc].*"/>
      <category regex="[Cc].*"/>
      <category regex=".*[Dd]atabase.*"/>
      <fixtures>
          <fileset dir="test-src">
              <include name="**/**.class"/>
          </fileset>
      </fixtures>
      <java failonerror="true">
          <classpath refid="project.class.path"/>
      </java>
  </jtiger>
  ```

## 6. Executing JUnit Test Cases
  ```

The JTiger framework provides the ability to execute test cases that have been written using the JUnit test framework. JUnit test cases typically inherit from the junit.framework.TestCase class, which is included with the JUnit test framework software. Although it is not mandatory for a class to inherit from junit.framework.TestCase for JTiger to execute it as a JUnit test case, it is mandatory that the class have setUp and tearDown methods that take no arguments. The class may have these methods through inheritance. That is, if a class inherits from junit.framework.TestCase (which declares the setUp and tearDown methods), it is not necessary to override them since it has setUp and tearDown methods through inheritance. If the setUp method does not exist when a class is executed by JTiger as a JUnit test case, all test cases in the test fixture will fail with a result of 'Failure (Set Up)'. If the tearDown method does not exist when a class is executed by JTiger as a JUnit test case, all test cases in the test fixture will fail with a result of 'Failure (Tear Down)'.

The JTiger framework provides the ability to execute JUnit test cases by providing two classes; org.jtiger.framework.junit.JUnitTestDefinition and org.jtiger.framework.junit.JUnitSetUpTearDown. The JUnitTestDefinition class must be used as the TestDefinition implementation and the JUnitSetUpTearDown class must be used a the SetUpTearDown implementation during test execution. The JTiger Apache Ant Task provides a shortcut for specfying these two classes by setting the "JUnit" attribute value to true.

## 7. Annotation Reference

#### 7.1 Overview

This section provides a reference for the annotation types that are included with JTiger. It is strongly recommended to use the JTiger Javadoc API Documentation alongside this reference during unit test development. All JTiger annotation types exist in the org.jtiger.framework package.

#### 7.2 Category

This annotation is used on test case methods or test fixture classes to specify one or more categories that test cases are in. If the annotation appears on a test fixture class, all test case methods in that fixture are in the categories specified as well as the categories that might be specified on each test case method itself.

```
@Category("Category1")
@Category(value = "Category1")
```

```
@Category({"Category1", "Category2"})
```

#### 7.3 ExpectException

This annotation is used on test case methods to indicate that an exception (any java.lang.Throwable or subclass) is expected to occur. If an expected exception does not occur, the test case results in failure.

```
@ExpectException(NullPointerException.class)
@ExpectException(value = NullPointerException.class)
@ExpectException(value = RuntimeException.class, subclass = true)
```

#### 7.4 Fixture

This annotation is used on test fixture classes to document the test fixture with information that appears in each test case result and JTiger reports. It is not mandatory for a test fixture class to use this annotation, but it is recommended since it improves the documentation of your test cases. If a test fixture is not annotated with @Fixture, the default name is used in the test case result, which is name of the class.

```
@Fixture("Fixture Name")
@Fixture(value = "Fixture Name")
@Fixture(value = "Fixture Name", description = "Fixture Description")
```

#### 7.5 Ignore

This annotation is used on test case methods to indicate that they should be ignored during test execution. This annotation is useful during test case development and debugging and it is considered a better practice to use an Ignore annotation than commenting out a test case. This is because an ignored test case will have a test case result of 'Ignored (Annotated)', where a commented test case is silently ignored and is prone to being forgotten.

```
@Ignore
@Ignore("This test case is ignored because I'm debugging")
@Ignore(value = "This test case is ignored because pizza arrived")
```

#### 7.6 Repeat

This annotation is used on test case methods to indicate that they should be repeated a specified number of times. Test case methods will have a distinct test case result for each time it is repeated. By default, this annotation has a value of one (1) indicating a single test execution. If a negative value is specified, the default value (1) is used.

```
@Repeat(7)
@Repeat(2L)
```

#### 7.7 SetUp

This annotation is used on test fixture methods to indicate that they should be executed prior to the execution of each test case method in the same test fixture class. If a method that has this annotation throws an exception when it is executed, the test case will result in 'Failure (Set Up)'. Set up methods are typically used to create test data that is used in test case methods.

```
@SetUp
```

#### 7.8 TearDown

This annotation is used on test fixture methods to indicate that they should be executed after execution of each test case method in the same test fixture class. If a method that has this annotation throws an exception when it is executed, the test case will result in 'Failure (Tear Down)'. Tear down methods are typically used to clean up resources that are used in the test case set up.

```
@TearDown
```

#### 7.9 Test

This annotation is used on fixture methods to indicate that they are test case methods and should be executed when the test fixture is executed. Use of this annotation implies that the test fixture will be executed using the default Test Definition implementation. If a test case method is annotated with @Test without a value attribute (the name of the test case method), the default name is used in the test case result, which is name of the method.

```
@Test
```

```
@Test("Test Case name")
@Test(value = "Test Case name")
@Test(description = "Test Case description")
@Test(value = "Test Case Name", description = "Test Case description")
```

# 8. The Self Test Fixtures

### 8.1 Overview

The JTiger download package includes source code that is used to assert that JTiger functions correctly and as specified. This test source code is based on the JTiger framework itself. JTiger development aims to achieve a certain metric of code coverage for unit test cases i.e. the test source code must execute a certain amount of the core JTiger code when the self test fixtures are run. That metric is 100% class and 100% method coverage.

The test source is a valuable resource in that it is a formal and very detailed specification of the requirements of JTiger. If there is a feature of JTiger that you are unsure of the exact specification of, even given the documentation for that feature, the test source code will almost certainly reveal the exact specification since the test source code utilises that feature (due to 100% method coverage).

### 8.2 The Test Report

The JTiger download package includes a report of the results of the JTiger test source code. A 100% success result indicates that JTiger is functioning correctly and meets all requirements, assuming that there are no defects in the requirements (the test source code). The test report is in HTML format and is generated by the class org.jtiger.report.html.HtmlFixtureResultsHandler. You can generate a report in the same format using this class as the configured Fixture Results Handler implementation class when you execute your test cases.

### 8.3 The Code Coverage Report

The JTiger download package includes a report of the amount of code that is covered during execution of the test source code. The report is generated by a code coverage tool called EMMA. The report indicates 100% class and 100% method coverage, which is an objective of the development of the JTiger software. This metric is intended to imply that JTiger is a robust and reliable unit test framework that can be used in any software production environment.

# 9. Example Scenario

### 9.1 Overview

This section provides an example scenario of using JTiger. The code under test in the example will be core J2SE classes. Since it is expected that a Java Runtime Environment includes core classes that meet all requirements, it is also expected that these test cases will succeed (have a test case result of 'Success'). The examples demonstrate a method of organization and documentation of test fixture classes. If you choose to adopt this method, it is merely consequential i.e. JTiger makes no recommendations about the organization of your test fixture classes.

### 9.2 Some Simple Test Fixtures

The following two test fixtures make assertions regarding the java.lang.Integer and java.lang.Thread class respectively.

- 
```
package org.jtiger.samples;

import static org.jtiger.assertion.Basic.assertEqual;
import static org.jtiger.assertion.Basic.assertTrue;
import org.jtiger.framework.Fixture;
import org.jtiger.framework.Category;
import org.jtiger.framework.Test;

@Fixture(value = "Thread", description = "Performs Unit Tests on
java.lang.Thread")
@Category({"java.lang.Thread", "J2SE Core", "java.lang"})
public final class SampleTestFixture1
{
    public SampleTestFixture1()
    {

    }

    @Test(value = "Constant fields", description = "Test the
constant fields of java.lang.Thread")
    public void constantFields()
    {
        assertEqual(1, Thread.MIN_PRIORITY);
        assertEqual(5, Thread.NORM_PRIORITY);
        assertEqual(10, Thread.MAX_PRIORITY);
    }

    @Test(value = "Runnable", description = "Test the Runnable
implementation passed to java.lang.Thread")
    public void runnable() throws InterruptedException
    {
        final MockRunnable r = new MockRunnable();
```

```
        final Thread t = new Thread(r);
        t.start();
        t.join();
        assertTrue(r.isRun());
    }

    private static final class MockRunnable implements Runnable
    {
        private boolean run;

        public void run()
        {
            run = true;
        }

        public boolean isRun()
        {
            return run;
        }
    }
}
```

- 

```
package org.jtiger.samples;

import static org.jtiger.assertion.Basic.assertEqual;
import static org.jtiger.assertion.Basic.assertSame;
import static org.jtiger.assertion.Serialization.assertSerializes;
import static
org.jtiger.assertion.Serialization.assertSerializesEqual;
import static
org.jtiger.assertion.Comparable.assertEqualComparesToZero;
import static
org.jtiger.assertion.ObjectFactoryContract.assertObjectFactoryFillsContract;
import static
org.jtiger.assertion.EqualsMethodContract.assertEqualsMethodFillsContract;
import static
org.jtiger.assertion.HashCodeMethodContract.assertHashCodeMethodFillsContract;
import org.jtiger.framework.Fixture;
import org.jtiger.framework.Category;
import org.jtiger.framework.Test;
import org.jtiger.framework.ExpectException;
import org.jtiger.assertion.ObjectFactory;

@Fixture(value = "Integer", description = "Performs Unit Tests on
java.lang.Integer")
@Category({"java.lang.Integer", "J2SE Core", "java.lang"})
public final class SampleTestFixture2
{
    public SampleTestFixture2()
    {

    }

    @Test(value = "Constant fields", description = "Test the
constant fields of java.lang.Integer")
    public void constantFields()
    {
        assertEqual(2147483647, Integer.MAX_VALUE);
        assertEqual(-2147483648, Integer.MIN_VALUE);
        assertEqual(32, Integer.SIZE);
        assertSame(int.class, Integer.TYPE);
    }
```

```
    @Test(value = "Constructors (1)", description = "Test the
constructors of java.lang.Integer")
    public void constructors1()
    {
        new Integer(7);
        new Integer("7");
        final Integer i1 = new Integer(7);
        assertEqual(7, i1.intValue());

        final Integer i2 = new Integer("8");
        assertEqual(8, i2.intValue());
    }

    @Test(value = "Constructors (2)", description = "Test the
constructors of java.lang.Integer")
    @ExpectException(NumberFormatException.class)
    public void constructors2()
    {
        new Integer("xyz");
    }

    @Test(value = "Rotate", description = "Test the rotating method
of java.lang.Integer")
    public void rotate()
    {
        final Integer i = 7;
        assertEqual(7340032, Integer.rotateRight(i, 12));
        assertEqual(28672, Integer.rotateLeft(i, 12));
    }

    @Test(value = "Serialization", description = "Test the
serialization of java.lang.Integer")
    public void serialization()
    {
        final Integer i = 7;
        assertSerializes(i, "Integer does not serialize");
        assertSerializesEqual(i, "Integer does not serialize
equal");
    }

    @Test(value = "Comparable", description = "Test the comparable
implementation of java.lang.Integer")
    public void comparable()
    {
        final ObjectFactory<Integer> factory = new
ObjectFactory<Integer>()
        {
            public Integer newInstanceX()
            {
                // WARNING!
                // Values between -128 and 127 will not work
                // here since they are boxed to the same
                // instance of Integer.
                // This violates the ObjectFactory contract.
                // This method must return unique instances on each
invocation.
                return 777;
            }

            public Integer newInstanceY()
            {
                return 888;
            }
```

```
        };

        assertObjectFactoryFillsContract(factory);
        assertEqualComparesToZero(factory, "equal Integers do not
compare to zero");
    }

    @Test(value = "Equals/HashCode", description =
        "Test the equals and hashCode method contract
implementations of java.lang.Integer")
    public void equalsHashCodeContracts()
    {
        final ObjectFactory<Integer> factory = new
ObjectFactory<Integer>()
        {
            public Integer newInstanceX()
            {
                return 777;
            }

            public Integer newInstanceY()
            {
                return 888;
            }
        };

        assertObjectFactoryFillsContract(factory);
        assertEqualsMethodFillsContract(factory);
        assertHashCodeMethodFillsContract(factory);
    }
}
```

#### 9.3 Executing the Test Fixtures From the Command Line

The following command line execution statement is intended to be used on a UNIX platform, since it uses a colon (:) as the path separator. If you are using a Microsoft Windows platform, substitute the colon with a semi-colon (;). The statement executes only test cases that are in the "java.lang" category. All of the test cases in the sample test fixtures are in this category, so it has no effect. If these test fixtures were accompanied by other test fixtures whose test cases were in other categories, this categorization would allow only a subset of test case methods to be executed during test execution. Note that the dot (.) is escaped with a backslash (\) since it has a special meaning in a regular expression and it is the literal meaning that is intended.

```
java -classpath tests:jtiger.jar org.jtiger.framework.FixturesRunnerMain
    -fixtureClasses org.jtiger.samples.SampleTestFixture1
org.jtiger.samples.SampleTestFixture2 -categories "java\.lang"
```

#### 9.4 Executing the Test Fixtures From the Apache Ant Task

The following is an Apache Ant task excerpt that executes the sample test fixtures using the JTiger Apache Ant Task:

```
<taskdef name="jtiger" classname="org.jtiger.ant.JTigerTask"
classpathref="project.class.path"/>
<jtiger haltonfailure="true">
    <category regex="java\.lang"/>
    <fixture classname="org.jtiger.samples.SampleTestFixture1"/>
    <fixture classname="org.jtiger.samples.SampleTestFixture2"/>
    <java failonerror="true"/>
</jtiger>
```

#### 9.5 Executing the Test Fixtures Using the API

The following is a code sample that runs the sample test fixtures and outputs a HTML test report of the test case results in a subdirectory of the user home directory using the JTiger API.

```
package org.jtiger.samples;

import static
org.jtiger.framework.FixturesRunnerConfigFactory.newFixturesRunnerConfig;
import static org.jtiger.framework.SequenceFactory.newSequence;
import static
org.jtiger.framework.FixturesRunnerFactory.newFixturesRunner;
import org.jtiger.framework.FixturesRunnerConfig;
import org.jtiger.framework.FixturesRunner;
import org.jtiger.framework.FixtureResults;
import org.jtiger.framework.RunnerException;
import org.jtiger.framework.FixtureResultsHandler;
import org.jtiger.framework.FixtureResultsHandlerException;
import org.jtiger.report.html.HtmlFixtureResultsHandler;
import java.io.File;

public final class SampleTestFixtureAPI
{
    private SampleTestFixtureAPI()
    {

    }

    public static void main(final String... args) throws
RunnerException, FixtureResultsHandlerException
    {
        final Class<?>[] fixtureClasses = new
Class<?>[]{SampleTestFixture1.class,
            SampleTestFixture2.class};

        final FixturesRunnerConfig config =
newFixturesRunnerConfig(newSequence(fixtureClasses),
            null, null, false, "java.lang");
        final FixturesRunner runner = newFixturesRunner();
        final FixtureResults results = runner.run(config);

        final FixtureResultsHandler handler = new
HtmlFixtureResultsHandler();
```

```
        final String dir = System.getProperty("user.home") +
"/.jtiger-sample-report";
        new File(dir).mkdirs();
        handler.handleResult(results, newSequence(new String[]{dir}));
    }
}
```

## 10. JTiger User Community

#### 10.1 IRC (JTiger live chat)

JTiger has an active IRC channel on the freenode network called #jtiger. Most IRC clients will connect to the freenode network and join the JTiger channel by executing the following IRC client command:

```
/server irc.freenode.net -j #JTiger
```

Most IRC clients also allow you to connect by following a URL. The URL for the JTiger IRC channel is irc://irc.freenode.net/JTiger. To get started with IRC on a Microsoft Windows platform, download and install mIRC and follow the JTiger IRC channel URL. If you are on a Linux platform, download and install xchat.

## 11. Future Direction for JTiger

JTiger future development is being investigated. Some of the features and topics that are under review include the development of a swing GUI interface from which to execute unit test cases, the development of IDE (Integrated Development Environment) plugins for Eclipse, and Intellij IDEA, and the possibility of a mock objects package that mocks J2SE and J2EE core classes.

Join the JTiger IRC channel and have your say! Make a suggestion, share your ideas or just hang out.

## Appendix A. Executing JTiger unit test cases using an Integrated Development Environment

JTiger currently does not include any IDE integration. However, this does mean that you cannot execute JTiger unit test cases while using the features of your

IDE. Since JTiger includes a main method from which to execute test cases, it is merely a matter of specifying the class name to the IDE along with the appropriate command line parameters. The class name is org.jtiger.framework.FixturesRunnerMain and typically you would pass one or more class names using the -fixtureClasses command line option.

Most IDEs allow you to specify a class name that contains the main method to execute along with the appropriate command line parameters. As an example, specify the class name "org.jtiger.framework.FixturesRunnerMain" and specify the value "-fixtureClasses org.jtiger.samples.SampleTestFixture1" as the program command line parameters. That's all there is to it - execute the application, put in debug breakpoints, and use the full set of features of your Integrated Development Environment.

## Appendix B. Credits

Thanks to Stephen Glass and David Moore for reviewing my work and giving me encouragement to keep fighting. Special thanks again to Dave for writing the XSL for the JTiger HTML reports.