UNIT TESTING

Written by Patrick Kua
Oracle Australian Development Centre
Oracle Corporation

TABLE OF CONTENTS

1 OVERVIEW

1.1 Document Purpose

This document was created to describe:

- What is testing?
- What is unit testing?
- The benefits of unit testing.
- The JUnit framework for testing of java programs.
- Best practices for JUnit.
- How to unit test BC4J components.
- Unit testing resources.

1.2 Target Audience

The target audience for this document includes any people involved in the software engineering process that have not witnessed the benefits of unit testing first-hand. People that are closely related to the development of code will benefit from this document the most. Even developers who write or who have written unit tests may benefit from the information contained in this document.

1.3 References

Note that most of the information in this document is widely available and that this document is intended to be a convenient place where all of this information is collated and presented. References to the various sources from which the information contained in this document has been based upon are given where applicable.

This document was heavily based on the Unit Testing Java Programs document composed by Brad Long and as of the writing of this document is available on Oracle Files Online in the "**Healthcare Development – Australia**" workspace.

2 TESTING

2.1 Introduction

Anyone who has been involved in any part of any software engineering project will have encountered some form of testing. It is a fundamental process that has traditionally been given the least priority or has been completed with the least amount of effort. There are countless textbooks, resources and websites devoted to this topic, but the purpose of testing generally remains the same. A generic definition for the term '**test**' as provided by Dictionary.com is "A procedure for critical evaluation; a means of determining the presence, quality, or truth of something; a trial".

Testing can therefore be considered an acceptance mechanism that demonstrates a piece of software is operating correctly as specified by its functional requirements. The majority of testing has generally been left late in the software engineering process, as outlined in the Waterfall Model[1]. Modern software engineering approaches such as Extreme Programming[2] take the opposite view by having a test-first strategy in which the amount of code developed is driven by the fulfilment of test cases that have been written first.

Regardless of the software engineering methodology that is adopted by your project, it is vital that some form of testing is completed.

2.2 Unit Testing

Testing comes in many types and forms, each of which may be used for different purposes and applied at different stages of a software engineering project. For example, functional testing focuses on validating that the main functions of a system fulfil their requirements, while system testing validates the most common usage paths of a system operate as expected prior to release. Performance testing on the other hand, is used to validate that the system meets its non-functional requirements by operating effectively under different load and stress conditions.

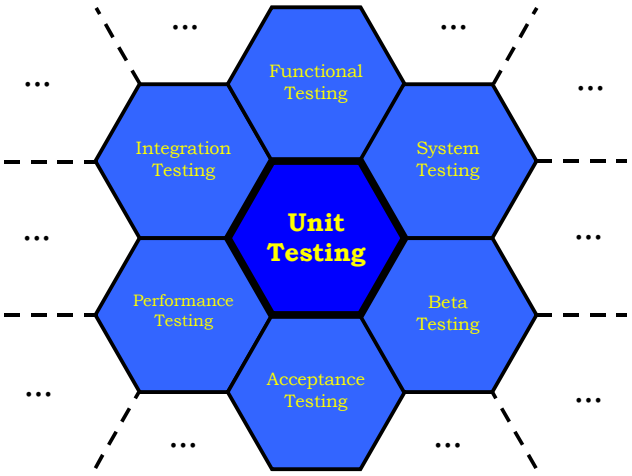

Figure 1: Comprehensive Unit Testing is a better foundation for other forms of testing

[1] The Waterfall Model is a model used to described the traditional approach to software engineer. - Managing the Development of Large Software Systems, IEEE 1970, Royce, Winston W

[2] Extreme Programming Website: http://www.extremeprogramming.org

In contrast, unit testing focuses on validating that the smallest unit in a system fulfils the contract that is defined by its API and documentation. Once each of these units have been tested and the tests are all passing, other forms of testing can be applied to validate that the software application meets its other requirements. In a diagrammatic form, unit testing can be considered the basis for many other forms of testing, as pictured in Figure 1.

From a practioner's perspective, unit testing java programs means writing a number of tests that validates the methods exposed by a class to other classes are operating as expected. The methods exposed by a class to other classes include public, protected and package-friendly methods.

2.3 Why Bother Unit Testing

There are many reasons why unit testing should be adopted for any software engineering project. These reasons include:

- **Faster Development** – As discussed in the JUnit Primer[3], 'When you write [unit tests] you'll spend less time debugging, and you'll have [more] confidence that changes to your code actually work. This confidence allows you to get more aggressive about refactoring code and adding new features. Without tests, it's easy to become paranoid about refactoring or adding new features because you don't know what might break as a result.'
- **Faster Debugging –** Without unit testing, the time it takes to debug or resolve a functional test that may have failed takes a long time to track down. With unit testing however, the scope of the test is kept to a minimum, and the point at which a failure may be triggered can be isolated faster.
- **Better Design and Documentation** – Writing unit tests forces developers into thinking how their code is going to be used and has generally resulted in better design and even better documentation. A comprehensive test suite can also help document the implicit behaviour of the methods exposed in the classes, providing another mechanism for developers to find out more about the intended design of the class.
- **Better Feedback Mechanism** – When all unit tests for a system are run as a whole, the state of the system as a whole can be measured. Unit tests also provide other developers with an instant mechanism of evaluating whether other parts of the code base are meeting their requirements or are still under development. Changes in test environments may sometimes cause problem with the code base, and continuous reporting of the unit test suites as a whole can help to indicate the state of the test environment.
- **Good Regression Tool** – A comprehensive unit test suite enables major refactorings and restructuring of the code base to be completed with greater confidence. As long as unit tests continue to pass, developers can be confident that their changes to the code do not have a negative impact on the functionality of the application as a whole.
- **Reduce Future Cost** – Many studies have proved that it costs significantly more money to fix a bug found later in its release than earlier in its development. A good unit test suite will uncover basic bugs early on in the development cycle, reducing the potential for other bugs and reducing cost of future maintenance.

[3] A JUnit Primer - http://www.clarkware.com/articles/JUnitPrimer.html#usage

2.4 Why Has Unit Testing Been So Neglected

There are many reasons why unit testing may not have been adopted for your software engineering project. These include:

- **Benefits not fully recognised or evaluated** – The benefits mentioned in the previous section may not have been identified and valued by members of the software engineering team. As a result, their software project misses out on the value-added proposition that unit testing offers.
- **No Time Was Allocated For Unit Testing** – Writing unit tests takes time for a developer to write. Thus it can be difficult to meet deadlines if time is not budgeted to include the time it takes to write unit tests as well as the code that meets the requirements.
- **Difficult to Write** – Before the JUnit[4] framework, there was no easy way for unit tests to be written in a manner that was easy, giving developers less of an incentive to write unit tests. Java's object-oriented nature also allows most classes to be tested in isolation, and Java's incremental compilation to class files makes changes to source and unit tests quick and easy to run.
- **Difficult to Manage** – The ad-hoc nature of which the unit tests had to be written prior to JUnit meant that projects had to develop their own standard for writing and managing their unit tests. Proprietary frameworks add to the time it takes for new developers to adjust to the project, while the lack of any unit testing framework makes it even more difficult for other developers to share the same code base project-wide.
- **Poor Development Attitude** - Some developers assume that their code will operate exactly the way they think the first time around. They might write a simple test harness or run through a few informal tests to validate their code operates, but rarely are these committed to any formal source control system. As a result, there is no guarantee that when changes are made to that code in the future that the intended behaviour does not change.

[4] JUnit – A unit-testing framework developed for testing Java classes. http://www.junit.org

3 JUNIT

3.1 Introduction to JUnit

JUnit is a testing framework that was developed for the Java programming language aimed at providing an easy way of developing unit tests. It is available from the JUnit website http://www.junit.org. One of the most favoured introductions to this framework is the JUnit Primer[5] article written by Mike Clark of Clarkware Consulting, Inc. An excellent description of JUnit's design is provided by the original authors Gamma and Beck in the Cook's Tour[6] available from the JUnit website.

3.2 Benefits of JUnit

The reasons that JUnit was adopted for the healthcare project include those reasons that Clark discusses in his article such as:

- **JUnit is a framework for unit testing** – All of the previously listed benefits of unit testing apply because JUnit is a framework for unit testing.
- **JUnit is elegantly simple** – The training time for developers to write unit tests is extremely short, and the structure of JUnit allows developers to start writing their own tests within minutes.
- **Writing JUnit tests is inexpensive** – Since training time is insignificant and the amount of code that is generally needed to unit test a particular class is quite small, the time consumed for developers is kept to a minimum, making the cost per unit test extremely low.
- **JUnit tests are written in Java** – Developers do not need to learn a different programming language in order to write unit tests, making the process that much easier. Its basis in java also allows for easy synchronisation between the tests and the main code base.
- **JUnit is opensource** – Resulting in no licensing costs to use the framework. Its opensource nature also means that many developers can and have contributed back to the code base to make it a better framework overall. It also gives developers access to the original source code, making it easy to customise the framework to support the unique requirements that may be needed for any software engineering project.
- **JUnit tests are easily managed** – JUnit allows an easy and simple way of managing unit tests, giving developers the ability to form a hierarchy within their unit tests. Forming a hierarchy enables developers to easily execute any number of unit tests that they want to execute at any point in time.

Other reasons that JUnit was adopted for the healthcare project include:

- **JUnit is becoming industry standard** – A standard testing framework not only allows new developers to quickly understand the way our unit tests are structured, and there are a number of extensions that are also being de facto standards for more specific types of unit testing.
- **JUnit is mature** – The testing framework has been through its early adopter's phase and is stable enough to use with confidence. There are also a number of resources widely available that detail its use and describe 'best-practices' that can be used for different types of applications or testing contexts.

[5] A JUnit Primer - http://www.clarkware.com/articles/JUnitPrimer.html

[6] Cook's Tour of JUnit - http://junit.sourceforge.net/doc/cookstour/cookstour.htm

- **Easy Integration With Ant** – The Ant[7] tool has been used heavily in healthcare to control the development process in a platform independent matter. JUnit is one of the standard optional tasks, making it even easier to execute the unit tests for different areas of the application.
- **JUnit is a generic testing framework** – Even though its original purpose was to facilitate unit testing, the stability and maturity of the framework allows JUnit to be leveraged for all other types of testing. Some examples include functional or system testing, integration testing and/or even performance testing.

3.3 JUnit Limitations

Many of JUnit's limitations are inherited from the limitations of unit testing in general. As a result, although JUnit is a generic test framework, especially well suited for unit testing, it does not work for every test environment. Built with a focus on providing a easy way of unit testing Java class methods, its ease-of-use and simplicity makes it impossible to apply to all situations effectively. Some of the limitations of JUnit (and unit testing in general) include the following.

- **Difficult to perform GUI tests effectively** – Testing interactions between Graphical User Interfaces and the components that define them proved very tedious and difficult in the past. JUnit does not provide a solution for testing GUI components very well.
- **Difficult to write tests for EJB components** – JUnit was designed in a simple manner to facilitate testing individual methods on a class that should be designed in a way that it is loosely coupled from other classes. Testing EJB classes in which several classes needed to be tested at the same time in different environments is not simple, and though it is possible to write tests for them using JUnit, they may not necessarily be tidy and simple to maintain.
- **Limited Reporting Mechanism** – JUnit's simplicity was designed to report only successes and failures for each unit test executed in a test suite. As a result, the framework does not automatically provide ways of maintaining complex statistics and analysis.
- **Time to Ramp Up** – Establishing a standard for unit testing and generating all the test stubs for each class can be a time consuming process. Although JUnit makes the process simple, it does not provide any easy way of automatically generating a number of test cases for a given piece of code.

There are a number of extensions or alternatives that address some JUnit's (and unit testing's) limitations. These are listed below:

- Artima (http://www.artima.com/suiterunner/index.html) - A runner that separates the notion of reporting with that of execution. This extension provides a replacement for JUnit's Graphical Test runner that allows multiple reports to be generated from the execution of a single test suite.
- Marathon Man (http://marathonman.sourceforge.net) - As described on their website, Marathon is a gui-test tool that allows you to play and record scripts against a java swing ui. It's written in java, and uses python as its scripting language - the emphasis being on an extremely simple, readable syntax that customers/testers/analysts feel comfortable with. It is full fledged python, so it is also extremely powerful and customisable for developer-types.
- JUnitDoclet (http://www.junitdoclet.org) - As described on their website, it lowers the step toward JUnit by offering a convenient way to create, organise and maintain JUnit tests. Generates TestSuites, TestCase skeletons and default tests from Java sources. Incremental behaviour keeps modified code when regenerating. Assists with refactorings (no tests get lost when renaming, moving, etc.). Because it is a Doclet it works very well with ANT.

[7] Ant, an open source build and task management system driven by XML - http://ant.apache.org/

- Cactus (http://jakarta.apache.org/cactus) - As described on the JUnit website, Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, etc). The intent of Cactus is to lower the cost of writing tests for server-side code. It uses JUnit and extends it. Cactus has been developed with the idea of automatic testing in mind and it provides a packaged and simple mechanism based on Ant to automate server-side testing. An alternative but complementary approach not covered by Cactus is to use Mock Objects.
- HttpUnit (http://httpunit.sourceforge.net) - Provides an easy mechanism for simulating HTTP requests/responses in order to test code that communications over HTTP.
- MockObjects (http://www.mockobjects.com) - A website devoted to building a framework for testing code with the use of mock objects. Mock objects can be used to replace complex resources in order to separate a resource dependency that may be encountered when testing complex systems.
- JMTUnit (http://www.michaelmoser.org/jmtunit) – A framework developed to help test multithreaded applications.

3.4 Developer's Introduction

This section will briefly introduce the main classes that will be used on a regular basis for any developers who may be writing unit tests. The very basic classes for JUnit are the Test interface, and the TestCase, TestSuite classes. The relationship between these classes are depicted in the diagram below:

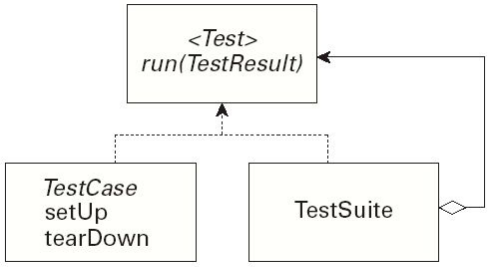

Figure 2: JUnit's class diagram for the three main classes

3.4.1 TestCase

The base class for all test classes, the **`TestCase`** provides a container for defining all unit tests. It implements the **`Assert`** interface providing an explicit way of stating the expected results of a unit tests. Some of the key methods in this class include:

- **`assertTrue`** – A method that provides a number of ways of ensuring a particular expression evaluates to true.
- **`assertEquals`** – A method that provide a number of ways of ensuring that two objects are equivalent.
- **`assertNull`** – A method to validate that a particular object is null.
- **`assertNotNull`** – A method to validate that a particular object is not null.
- **`fail`** – A method to force a failure when it is reached. This method is intended for use to flag branches in code that should not have been reached. A typical use for this statement applies when an expected exception has not been thrown.

All of the assertions above have overloaded methods that allow developers to specify a meaningful message that will be displayed if the assertion fails.

The TestCase class also provides two methods for maintaining the environment required for each of the test cases defined. These methods are:

- `setUp()` – The code defined in this block of code is called **before** each test is run to provide for hooks that set up the environment or initialise objects for every test. This method is only called exactly **once** before every test **method**.
- `tearDown()` – The code that is defined in this block of code is called **after** each test is run to provide for hooks that clean up the test environment. This method is only called exactly **once** after every test **method**.

3.4.2 TestSuite

The `TestSuite` class does not provide much functionality but what it does provide is a simple mechanism for managing tests. Each test suite enables individual tests, or a suite of other tests to be embedded in itself. The recursive manner in which test suites can be embedded in other test suits allows for a hierarchy of unit tests to be defined easily.

3.4.3 TestDecorator

The `TestDecorator` class is part of the extensions to the junit framework. These classes provide additional classes that are not considered essential to writing unit tests, but may provide useful functionality in writing tests. The javadoc from JUnit describes this class ‘as the base class for defining new test decorators. Test decorator subclasses can be introduced to add behaviour before or after a test is run.’

3.4.4 TestSetup

The `TestSetup` class extends the `TestDecorator` class and its main purpose is to act as the base for an anonymous test class that allows more control over the `setUp` and `tearDown` at levels higher than each test method.

Its most useful application is trying to improve the time of execution for all unit tests by establishing the testing environment once at the beginning of a test suite, instead of before and after each unit test.

3.4.5 Failures vs Errors

JUnit provides for two indicators that a test is not passing (failures and errors). Although the end result is the same, it is sometimes useful to be able to distinguish between the two.

Failures - A failure can be considered the result of an anticipated error that a developer has made provisions for. These are triggered when assertions fail, or when a fail is called programmatically. A failure generally indicates that a piece of code is no longer exhibiting the same functional behaviour.

Errors - An error can be considered an unanticipated error that has caused the test to fail. This means that an exception was thrown from the test method and was not caught programmatically by the unit test developer. An error generally gives more of an indication that larger problems may be causing the behaviour of the system to fail. These can include things such as changes to the test environment or the lack of an available resource.

4 AN EXAMPLE

The basics steps involved with creating a unit test are fairly straightforward:

1. Define a test class that will hold unit tests
2. Initialise all objects needed for the unit test
3. Call operations on the objects involved in the unit test, stating what assertions or failures are expected as result of the operations called.
4. Tear down the testing environment.
5. Execute the unit tests

4.1 Class to Test

The following is the code listing for a simple class that has been generated by a developer. The following block of code defines the class and method that we are going to test. The complete code listing for both the test class is given in the appendix.

```
package oracle.apps.example;

import java.io.Serializable;
import java.util.Date;

/**
 * This class is a simple test class that is intended to demonstrate how to
 * construct a unit test to test the methods contained in this class.
 *
 * An instance of this class represents a count of the number of hits a web
 * page may have. Each page may or may not have its own hit counter. A hit
 * counter stores an internal number of hits that have been recorded, and
 * the date at which the hit counter was first created.
 */
public class HitCounter implements Serializable
{
  private int numberOfHits;
  private Date creationDate;

  /**
   * Construct a new counter that does not have any hits as yet.
   */
  public HitCounter()
  {
    creationDate = new Date();
    numberOfHits = 0;
  }

  /**
   * Returns the number of hits that this HitCounter has currently
   * observed.
   *
   * @return The number of hits that this Hitcounter has observed.
   */
  public int getNumberOfHits()
  {
    return numberOfHits;
  }

  /**
   * Returns the date that this HitCounter was created.
   *
   * @return The date that this HitCounter was created.
   */
  public Date getCreationDate()
  {
    return creationDate;
  }

  /**
   * Triggers an increment in the number of hits held by this HitCounter.
   */
  public void incrementNumberOfHits()
  {
    numberOfHits ++;
  }
```

```
  /**
   * Generate a unique hash integer for this HitCounter.
   *
   * @return a number that is a unique hash code for this HitCounter.
   */
  public int hashCode()
  {
    int result;
    result = numberOfHits;
    result = 29 * result + creationDate.hashCode();
    return result;
  }

  /**
   * Implementation of the equals method for a HitCounter. A
   * hit counter can be considered the same if and only if the
   * number of hits and the creation date of the hit counters
   * are the same.
   *
   * @return <code>true</code> if the object is considered the same as this
   *         HitCounter, otherwise <code>false</code>
   */
  public boolean equals(Object object)
  {
    if ( object == null ) { return false; }
    if ( this == object ) { return true; }
    if ( !(object instanceof HitCounter) ) { return false; }

    final HitCounter hitCounterToCompare = (HitCounter) object;
    if (numberOfHits != hitCounterToCompare.numberOfHits) { return false; }
    if (!creationDate.equals(hitCounterToCompare.creationDate))
    {
      return false;
    }

    return true;
  }

  /**
   * Implementation of the standard toString() method, resulting
   * in a string that describes the current values of this instance
   * of the HitCounter.
   *
   * @return A string that represents a description of the current
   *         values of this HitCounter.
   */
  public String toString()
  {
    return "HitCounter{" +
           "numberOfHits=" + numberOfHits +
           ", creationDate=" + creationDate +
           "}";
  }
}
```

4.2 Generate the TestClass

The first step involves subclasses the TestCase class that will hold all the unit tests for testing this class. See the code listing below:

```
package oracle.apps.example;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.textui.TestRunner;

/**
 *
 */
public class HitCounterTest extends TestCase                    (A)
{
  public HitCounterTest(String testCaseName)                    (B)
  {
    super(testCaseName);
  }

  public static Test suite()                                    (C)
  {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(HitCounterTest.class);
    return suite;
  }
```

```
  public static void main(String[] args)                          (D)
  {
    TestRunner.run(suite());
  }
}
```

(A) Any class that is expected to run as a test must have the **`TestCase`** class as its superclass.

(B) All subclasses must override this constructor method that is defined by the **`TestCase`** class. The **`testCaseName`** represents the name of a method defined in this class that represents a unit test.

(C) The suite method provides a convenient method of automatically keeping the suite of tests for this class up to date. JUnit will automatically treat any methods in this class that start with **`test`** as a test method and add them to the test suite.

(D) Adding the main allows for any developer to automatically run all tests defined in the suite method for this class. There are a number of different runners, for this example, we are using a runner that produces results in a textual format.

4.3 Initialise objects needed for the unit test

Since we are writing tests for the HitCounter class, we will need to have access to access to an instance of the class so we can test the instance variables. The following block of code adds in the instance and sets up the data ready for testing.

```
.,.
public class HitCounterTest
{
  private HitCounter hitCounter;

  ...

  /**
   * Override the setUp method here to initialise objects that are tests
   * need.
   */
  protected void setUp()
  {
    hitCounter = new HitCounter();
  }

  ...
}
```

4.4 Writing the unit tests

4.4.1 Testing the incrementNumberOfHits() method

The increment method is easy to test because it is not too complex. There are no cases where exceptions are expected, so a single test case was generated to validate its behaviour operates effectively. The test below makes sure that the method only increments the counter by 1 each time it is called.

```
...

/**
 * Tests that the increment number of hits method on the HitCounter
 * class is operating as expected.
 *
 * When we effectively 'hit' the HitCounter, we should expect to
 * increment exactly the number of times we hit it.
 */
public void testIncrementMethod()
{
   final int CURRENT_NUM_OF_HITS = hitCounter.getNumberOfHits();
   final int NUMBER_OF_HITS = 10;
   for ( int i = 0; i < NUMBER_OF_HITS; i++ )
   {
     hitCounter.incrementNumberOfHits();
   }
   assertEquals("Number of expected hits was different",
                CURRENT_NUM_OF_HITS + NUMBER_OF_HITS,
                hitCounter.getNumberOfHits() );
}

...
```

4.4.2 Testing the toString() method

The **`toString()`** method is the one method that is generally untested unless a custom **`toString()`** is actually written by a developer. The following code simple validates that the **`toString()`** method produces a standard formatted string describing the values contained in the instance of **`HitCounter`** that it is called on.

```
...

/**
 * This validates that the toString() method is operating correctly.
 * toString() should produce a useful debugging string that details
 * the values of its fields.
 */
public void testToString()
{
  final int CURRENT_NUM_OF_HITS = hitCounter.getNumberOfHits();
  final Date CREATION_DATE = hitCounter.getCreationDate();
  final String EXPECTED_STRING = "HitCounter{numberOfHits=" +
                                 CURRENT_NUM_OF_HITS + ", creationDate=" +
                                 CREATION_DATE + "}";
  assertEquals("toString() method does not produce the expected output",
               EXPECTED_STRING, hitCounter.toString() );
}

...
```

4.4.3 Testing the equals() method

The **`equals()`** method must meet several conditions in order for it to fulfil its contract. These include:

- A call to equals with a null parameter will return false.
- A call to equals given its own reference should always return true.
- If all attributes for the object are the same values and it is an instance of the same class, equals should return true.

Testing equality when using different references is sometimes a little complex. In this example for instance, we cannot modify the **`creationDate`** attribute as the constructor is responsible for initialising this attribute. To get around this problem, the following piece of code was added to the test class to provide an exact copy of the object we are testing through serializing and deserializing the object in memory to force java to give us a different reference to the object. Depending on actual usage requirements, developers may find it better to refactor this into the superclass, or into a separate class altogether.

```
...

/**
 * Clones an object via serialization. The object will be written to an
 * in memory output stream, and read back in from that stream and
 * reconstructed into the object that it should have been.
 *
 * @param serializableObject An object that is serializable
 * @return A copy of the object where all serializable fields have the
 *         same value as that of the original.
 * @throws IOException If there is a problem writing the object
 * @throws ClassNotFoundException If the object passed in does not have
 *         its class definition loaded into memory.
 */
protected Object cloneViaSerialization( Object serializableObject )
  throws IOException, ClassNotFoundException
{
  // Serialize the object to memory
  ByteArrayOutputStream inMemoryOutputStream =
    new ByteArrayOutputStream();
  ObjectOutputStream serializer =
    new ObjectOutputStream(inMemoryOutputStream);
  serializer.writeObject(hitCounter);
  serializer.flush();

  // Read the object out from memory
  ByteArrayInputStream inMemoryInputStream =
    new ByteArrayInputStream(inMemoryOutputStream.toByteArray() );
  ObjectInputStream deserializer =
    new ObjectInputStream(inMemoryInputStream);
  return deserializer.readObject();
}

...
```

The actual test methods for testing the **`equals()`** method is shown below:

```
...

/**
 * Test that a null passed into the equals() method returns false.
 */
public void testEqualsWithNullValue()
{
   assertTrue("equals(null) does not result in a false boolean value",
              !hitCounter.equals(null) ); // note the ! symbol
}

/**
 * Test that validates that a call to the equals(Object) method with
 * the same reference results in true.
 */
public void testEqualsWithSameReference()
{
  assertTrue("equals(this) does not result in a true boolean value",
             hitCounter.equals(hitCounter) );
}

/**
 * Create an object that captures exactly the same data as the
 * original hit counter, but are different references. Make sure
 * that the equals() method returns true for the comparison
 */
public void testEqualsWithPerfectClone() throws Exception
{
  HitCounter copyOfHitCounter =
    (HitCounter)cloneViaSerialization(hitCounter);
  assertTrue("equals() method of hitcounter should return true with a " +
             “copy of a hitcounter",
             hitCounter.equals(copyOfHitCounter));
  assertTrue("equals() method should work in reverse",
             copyOfHitCounter.equals(hitCounter));
}

...
```

4.4.4 Testing the hashCode() method

Similar to the **`equals()`** method, the unit tests written for testing the **`hashCode()`** function leverage the **`cloneViaSerialization(Object)`** method to create copies of a class.

```
...

/**
 * Validate that the hash code for a hit counter and its perfect
 * clone is the same.
 */
public void testHashCodeWithPerfectClone() throws Exception
{
  HitCounter copyOfHitCounter =
     (HitCounter)cloneViaSerialization(hitCounter);
  assertEquals("Hashcode for the hit counter and its perfect clone " +
               "should be the same",
               hitCounter.hashCode(), copyOfHitCounter.hashCode() );
}

/**
 * Validate that the hash code is generated consistently regardless
 * of the time that passes.
 */
public void testHashCodeIsConsistent() throws Exception
{
  assertEquals("HashCode generated for the same object should be " +
               "consistent",
               hitCounter.hashCode(), hitCounter.hashCode() );
}

...
```

4.5 Running the unit tests

Once a test suite has been completed, JUnit provides a number of methods for executing a test suite or any of its tests cases individually. Running a test suite will automatically run all of its subordinate **`TestCase`** instances and **`TestSuite`** instances that were nested inside. Running a **`TestCase`** class will automatically invoke all of its public **`testXXX()`** methods.

JUnit provides both a textual and a graphical user interface. Both user interfaces indicate how many tests were run, any errors or failures, and a simple completion status. The simplicity of the user interfaces is the key to running tests quickly. You should be able to run your tests and know the test status with a glance, much like you do with a compiler.

The textual user interface (**`junit.textui.TestRunner`**) displays "OK" if all the tests passed and failure messages if any of the tests failed. For example, running the test class just written, we see the following screen:

Figure 3: Running the main() method on the test class

We can also use the use the graphical user interface test runner (**`junit.swingui.TestRunner`**) that displays a Swing window with a green progress bar if all the tests passed or a red progress bar if any of the tests failed. To run all the tests inside the specified class, we run the following piece of code:

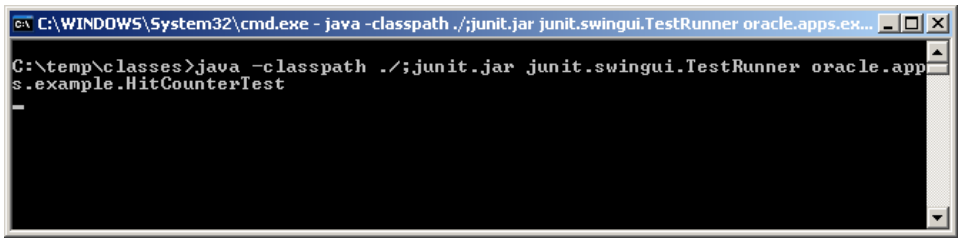

Figure 4: Running the GUI TestRunner

This results in the following interface being displayed on a windows machine.

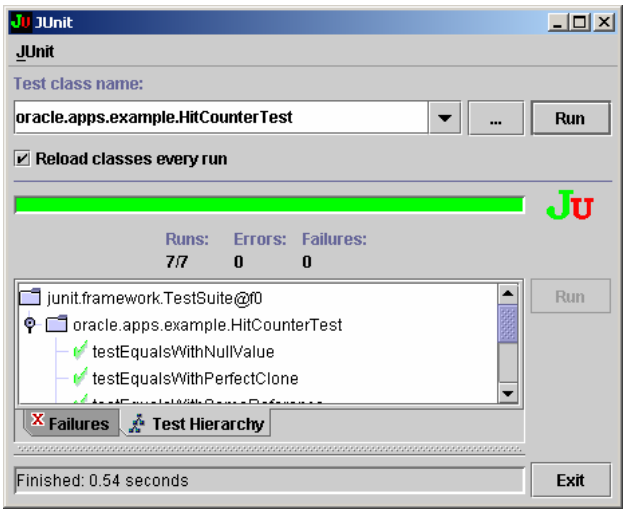

Figure 5: The GUI test runner interface

In general, **`TestSuite`** and **`TestCase`** classes should define a **`main()`** method which employs the appropriate user interface. The tests we've written so far have defined a **`main()`** method employing the textual user interface.

5 BEST PRACTICES

This section will discuss how to effectively create and manage your unit tests. It will recommend various strategies for unit testing and how to get the most out of JUnit.

5.1 Setting up the Test Environment

5.1.1 Test Classes should be placed in an appropriate directory

Test code is generally separated from source code so that both can be individually built and distributed in different manners. Separating them also helps the development process for clearly segregating test and source code. The most widely accepted structure is to have the same package structure in a different directory from the source code and have test classes reside in the mirrored package structure under the test source. This allows the test class to access protected or package-friendly (default access modifier) methods, but still maintains the separation between source and test code. Any number of test base directories can also be created depending on the types of tests that are being written.

This same approach should be adopted for the actual output directory of the classes with the following things in mind:

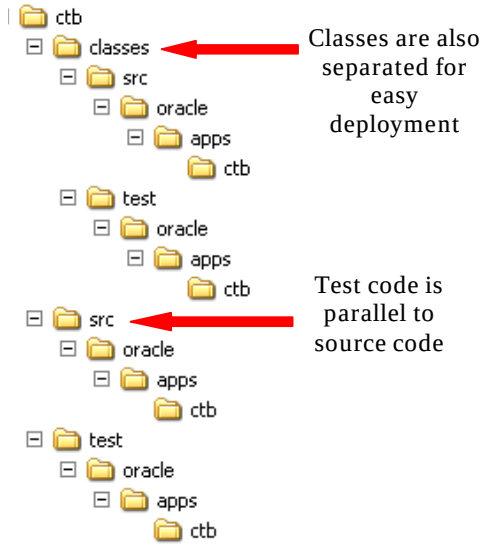

Figure 6: Recommended project structure

5.1.2 Define a standard base TestClass

Tests written to test the same application may perform the same sort of operations over and over. Sometimes, tests may need to acquire a resource in a complex manner, but need to do so for every test in a common manner (for example, gaining access to Application Modules), so it is recommended that you define a base class for all tests in your system so common methods can be refactored into the superclass when needed.

Common methods applicable unit testing wide allows for a convenience place holder to hold methods like these. Depending on the particular circumstance and number of common methods, it might also be better to refactor these methods into a separate package where needed for better object-oriented design.

5.1.3 Define a standard naming convention for Test Classes

In many circumstances it is useful to be able to distinguish between classes that represent the code being tested, the code that is testing it, and the code that organises the tests. A default naming convention that may be adopted includes the following:

- Leave source code class names with their original names.
- The class that is testing the original class should be named of the same name, but appended with **`Test`**.
- The class that organises tests should have **`TestSuite`** appended to it.

The naming conventions will also help in executing the different suites from Ant as it can recursively call the unit testing target against a number of classes that match a specified pattern. An example naming pattern is depicted below:

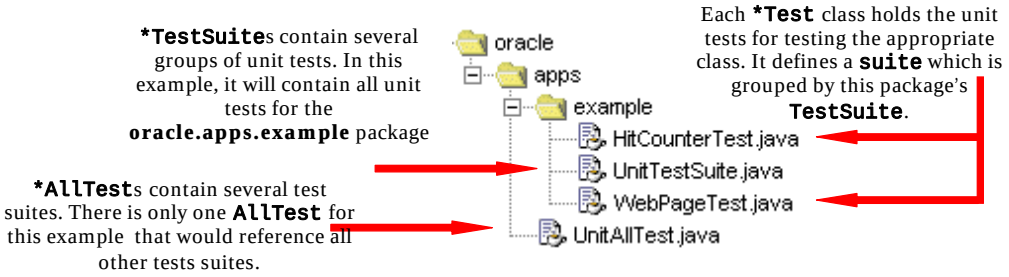

Figure 7: An example naming convention

5.1.4 Make the process of writing tests easy

Sometimes the process of writing unit tests can be difficult depending on the types of unit tests being written. Thus it is important that the unit testing environment is created to facilitate writing unit tests. Part of this process might be defining a coding standard that is to be followed when writing unit tests. Another part might be a training manual or course that introduces new developers to the composition of the current unit test suite, or even a mailing list or bulletin board that communicates changes or recommendations to developers when writing unit tests.

An important, but fairly neglected part of this process should also make provisions for constant refactoring to be applied to the unit test code. A result of the refactoring process on the test code will result in a more manageable test suite and perhaps even a set of standard assistant classes that encapsulates a group of functionality that is difficult to write in a single unit test cleanly. The functions that these assistant classes might provide include:

- Executing a specified SQL script against a particular database (i.e. the one that an application will be running on).
- Cloning of objects with a different reference (like the serialization/deserialization example previously given).
- Automatic generation of comparators for standards objects (highly useful when the valueobject pattern is applied).
- Separation of common set up that might need to be run to establish a common unit test environment.

5.2 Writing Tests

5.2.1 Develop a unit test for every Java class (where possible)

A unit test should exist for every Java class that has been manually modified by the development group. It is difficult to write unit tests for some classes due to the nature of that component (such as UI) so it may not be feasible to write comprehensive unit tests for those aspects. But to quote Martin Fowler, “It is better to write and run incomplete tests than not to run any complete tests.”

It is more important to write unit tests for code generators than it is for the code that was generated. Maintaining unit tests for code that may change at any instant will prove unmanageable over time and will not indicate where the problem originates.

5.2.2 Define tests correctly

Naming of test methods is important to communicate to other users the immediate purpose of the test. If the test cannot be succinctly named, then it is important to document the purpose of the test in the javadoc. All test methods should follow the pattern below:

```
public void testXXX() [throws Exception]
```

All test methods should be declared as **public**, and have **no return value**. It is also recommended that all tests methods be prepended with the term **`test`** for several reasons. These include:

- Test methods can immediately be distinguished from all other methods that may be defined in the test class.
- JUnit has the ability to add all tests in a class to a test suite, and uses introspection to find all methods that start with ‘test’. This is a feature added to the framework to make it easy when composing test suites.
- It is generally accepted practice to name all test methods this way.

An example of well-defined unit test methods is given below.

```
public class SomeClass extends TestCase
{
  ...

  public void testSomeMethodWithNegativeInput()
  {
    ...
  }

  ...

  public void testSomeMethodWithNullInput()
  {
    ...
  }

  /**
   * Returns a TestSuite that contains all of the methods that start
   * start with the word test that are defined in this class.
   */
  public static Test suite()
  {
     TestSuite suite = new TestSuite(“Some class’ test suite”);
     suite.addTestSuite(SomeClass.class);
     return suite;
  }

  ...

}
```

It is also recommended that developers steer clear of ‘de-vowelling’ or writing non-descriptive test methods names as these are useless to other developers in communicating the intention of the test method. Example of test method names to avoid include: testScenario1, testSmMtdNull, testBlah, etc. It is much better to provide a more-descriptive method name, even if it may be appear too long.

5.2.3 Do not use the test-case constructor to set up a test case

Extracted from Javaworld's "JUnit best practices" article[8].

Setting up a test case in the constructor is not a good idea. Consider

```
public class SomeTest extends TestCase
{
   public SomeTest (String testName)
   {
      super (testName);
      // Perform test set-up
   }
}
```

Imagine that while performing the setup, the setup code throws an IllegalStateException. In response, JUnit would throw an AssertionFailedError, indicating that the test case could not be instantiated. The stack trace that is generated proves rather uninformative; it only indicates that the test case could not be instantiated. It doesn't detail the original error's location or place of origin. This lack of information makes it hard to deduce the exception's underlying cause.

Instead of setting up the data in the constructor, perform test setup by overriding setUp(). Any exception thrown within setUp() will thus be reported correctly.

5.2.4 Don't assume the order in which testing within a test case run

Extracted from Javaworld's "JUnit best practices" article[9].

You should not assume that tests will be called in any particular order. Consider the following code segment:

```
public class SomeTestCase extends TestCase
{
   public SomeTestCase (String testName)
   {
      super (testName);
   }
   public void testDoThisFirst ()
   {
     ...
   }
   public void testDoThisSecond ()
   {
     ...
   }
}
```

In this example, it is not certain that JUnit will run these tests in any specific order when using reflection. Running the tests on different platforms and Java VMs may therefore yield different results, unless your tests are designed to run in any order. Avoiding temporal coupling will make the test case more robust, since changes in the order will not affect other tests. If the tests are coupled, the errors that result from a minor update may prove difficult to find.

In situations where ordering tests makes sense -- when it is more efficient for tests to operate on some shared data that establish a fresh state as each test runs -- use a static suite() method like this one to ensure the ordering:

```
public static Test suite()
{
   suite.addTest( new SomeTestCase("testDoThisFirst") );
   suite.addTest( new SomeTestCase("testDoThisSecond") );
   return suite;
}
```

[8] JUnit best practices, JavaWorld

[9] JUnit best practices, JavaWorld

There is no guarantee in the JUnit API documentation as to the order your tests will be called in. JUnit's current implementation employs a Vector to store tests so you can expect the above tests to be executed in the order they were added to the test suite. However this current implementation should not be relied upon.

5.2.5 Avoid writing test cases with side effects

The previous best-practice can be built upon with this one extracted from Javaworld's "JUnit best practices" article[10].

Test cases that have side effects exhibit two problems:

- They can affect data that other test cases rely upon
- You cannot repeat tests without manual intervention

In the first situation, the individual test case may operate correctly. However, if incorporated into a TestSuite that runs every test case on the system, it may cause other test cases to fail. That failure mode can be difficult to diagnose, and the error may be located far from the test failure.

In the second situation, a test case may have updated some system state so that it cannot run again without manual intervention, which may consist of deleting test data from the database (for example). Think carefully before introducing manual intervention. First, the manual intervention will need to be documented. Second, the tests could no longer be run in an unattended mode, removing your ability to run tests overnight or as part of some automated periodic test run.

5.2.6 Ensure that tests are time-independent

Extracted from Javaworld's "JUnit best practices" article[11].

Where possible, avoid using data that may expire; such data should be either manually or programmatically refreshed. It is often simpler to instrument the class under test, with a mechanism for changing its notion of today. The test can then operate in a time-independent manner without having to refresh the data.

5.2.7 Leverage JUnit's Assertions and Exception Handling

Extracted from Javaworld's "JUnit best practices" article[12].

Many JUnit novices make the mistake of generating elaborate try and catch blocks to catch unexpected exceptions and flag a test failure. Here is a trivial example of this:

```
public void exampleTest ()
{
  try
  {
    // do some test
  }
  catch (SomeApplicationException e)
  {
    fail ("Caught SomeApplicationException exception");
  }
}
```

JUnit automatically catches exceptions. It considers uncaught exceptions to be errors, which means the above example has redundant code in it.

---

[10] JUnit best practices, JavaWorld

[11] JUnit best practices, JavaWorld

[12] JUnit best practices, JavaWorld

Here is a far simpler way to achieve the same result:

```
public void exampleTest () throws SomeApplicationException
{
  // do some test
}
```

In this example, the redundant code has been removed, making the test easier to read and maintain (since there is less code). In addition to this, the stack trace produced from the error is available instead of just the failure message that tells us that the test fails.

Use the wide variety of assert methods to express your intention in a simpler fashion. Instead of writing:

```
assert (creds == 3);
```

Write:

```
assertEquals ("The number of credentials should be 3", 3, creds);
```

The above example is much more useful to a code reader. And if the assertion fails, it provides the tester with more information. JUnit also supports floating point comparisons:

```
assertEquals ("some message", result, expected, delta);
```

When you compare floating point numbers, this useful function saves you from repeatedly writing code to compute the difference between the result and the expected value.

Use assertSame() to test for two references that point to the same object. Use assertEquals() to test for two objects that are equal.

5.2.8 Keep tests small and fast

Extracted from Javaworld's "JUnit best practices" article[13].

Executing every test for the entire system shouldn't take hours. Indeed, developers will more consistently run tests that execute quickly. Without regularly running the full set of tests, it will be difficult to validate the entire system when changes are made. Errors will start to creep back in, and the benefits of unit testing will be lost. This means stress tests and load tests for single classes or small frameworks of classes shouldn't be run as part of the unit test suite; they should be executed separately.

5.2.9 Document tests in javadoc

Extracted from Javaworld's "JUnit best practices" article[14].

Test plans documented in a word processor tend to be error-prone and tedious to create. Also, word-processor-based documentation must be kept synchronized with the unit tests, adding another layer of complexity to the process. If possible, a better solution would be to include the test plans in the tests' javadoc, ensuring that all test plan data reside in one place.

5.2.10 Avoid manual intervention

Removing the need for any intervention by developers is a good key to writing unit tests. Debug statements may help when trying to track down a particular problem, but a good unit test suite immediately narrow down the area where debugs statements are needed when tests fail.

The standard debugging statements `System.out.println` and `System.err.println` should be removed at all costs as they can be rewritten into assertions much easier. A trivial example of introducing manual intervention is demonstrated below:

---

[13] JUnit best practices, JavaWorld

[14] JUnit best practices, JavaWorld

```
public void testMethodManually()
{
  final String ORIGINAL_STRING = “MyString”;
  String convertedString = MyCode.formatStringUpper(ORIGINAL_STRING);
  System.out.println(“Converted string is : “ + convertedString);
}
```

This can easily be rewritten as follows:

```
public void testMethodAutomatically()
{
  final String ORIGINAL_STRING = “MyString”;
  final String EXPECTED_STRING = “MYSTRING”;

  String convertedString = MyCode.formatStringUpper(ORIGINAL_STRING);
  assertEquals(“Converted string was not expected, EXPECTED_STRING,
               convertedString );
}
```

This not only removes the manual intervention required to execute the test completely, but it also further demonstrates the intention of the test to other people who may read the code.

The following was extracted from Javaworld’s “JUnit best practices” article[15].

Testing servlets, user interfaces, and other systems that produce complex output is often left to visual inspection. Visual inspection -- a human inspecting output data for errors -- requires patience, the ability to process large quantities of information, and great attention to detail: attributes not often found in the average human being. Below are some basic techniques that will help reduce the visual inspection component of your test cycle.

- Swing - When testing a Swing-based UI, you can write tests to ensure that:
  - All the components reside in the correct panels
  - You've configured the layout managers correctly
  - Text widgets have the correct fonts
  - Basically all GUI widgets have the values of attributes set as expected. In other words, you can test the model underlying the view of the components.
- XML
  - When testing classes that process XML, it pays to write a routine that compares two XML DOMs for equality. You can then programmatically define the correct DOM in advance and compare it with the actual output from your processing methods
- Servlets
  - With servlets, a couple of approaches can work. You can write a dummy servlet framework and preconfigure it during a test. The framework must contain derivations of classes found in the normal servlet environment. These derivations should allow you to preconfigure their responses to method calls from the servlet.

You can avoid visual inspection in many ways. However, sometimes it is more cost-effective to use visual inspection or a more specialised testing tool. For example, testing a UI's dynamic behaviour within JUnit is complicated, but possible. It may be a better idea to purchase one of the many UI record/playback testing tools available, or to perform some visual inspection as part of testing. However, that doesn't mean the general rule -- don't visually inspect -- should be ignored

[15] JUnit best practices, JavaWorld

5.2.11 Catch the most specific exception

When writing tests for situations in which an exception is expected, it is extremely easy to write a test that covers up other potential errors. Consider the following block of code:

```
public void testFileReaderWithNullInput()
{
  try
  {
    mainClass.parseFile(null);
    fail("An exception should have been generated with a null input");
  }
  catch ( Exception exception )
  {
     // exception expected
  }
}
```

or worse yet:

```
public void testFileReaderWithNullInput()
{
  try
  {
    mainClass.parseFile(null);
    fail("An exception should have been generated with a null input");
  }
  catch ( Throwable throwable )
  {
     // exception expected
  }
}
```

Although the parsing of a file with a null input is expected to throw an exception, the real problem is that the two unit tests cover up any other exceptions or bugs that may be thrown by the code. The last catch block in the above examples will catch all other types of exceptions that are thrown, so it is better to be as specific as possible. The improved test case can be rewritten as below:

```
public void testFileReaderWithNullInput() throws IOException
{
  try
  {
    mainClass.parseFile(null);
    fail("An exception should have been generated with a null input");
  }
  catch ( NullPointerException nullPointerException )
  {
     // nullPointerException expected
  }
}
```

Not only do we catch the `NullPointerException` that is expected, but throw the `IOException` that the method declares, keeping the unit test clean, simple and readable.

5.2.12 Add at least one test case for every bug exposed

When a bug is logged against a system, it indicates that the behaviour of the system is not operating correctly. A test case can be considered a way of defining the functionality of a system by capturing in a number of assertions/failures the intended response. Thus, when a bug is defined, it means that a test case should exist that should be failing.

If there was no unit test, then one should be written. Writing a unit test for the bug will not only clarify what is causing the bug, and perhaps contribute to a better redesign of that sub-system, but once fixed, will act as a monitor for that piece of code. If it fails to fulfil its contract with other parts of the system, then developers will be notified that it has failed to do so.

5.2.13 Tweak unit tests for performance only when needed

As the number of unit tests grow, it is important that the entire test suite is runnable for all developers within an acceptable amount of time. The initial stages involved with testing for a project will normally be focused on establishing the standards for your unit testing framework to making it easier for developers to write tests.

Due to the variety of different applications, it is difficult to design standards that guarantee performance, so the performance tuning of unit tests should only be looked at when it is operating at unacceptable levels. Apply different strategies for the different bottlenecks that may be causing unit tests to fail. The most common is having setUp and tearDown methods that do too much. Some example strategies that could be considered include:

- Extracting common set up and tear down code to be executed once off before a series of tests that all use the same things. An example of this code is given in Appendix C.
- Loading pre-serialized objects into memory for instant comparable states instead of querying them from a database.
- Moving tests to be executed on a faster machine.
- Pre-seeding data into a database.
- Use of mock objects to replace external resources.

5.2.14 If it's too hard to test, perhaps it's hard to use – Refactor

Writing unit tests for some pieces of code is inherently difficult, but more often than not, unit tests should be fairly simple to write. A unit test demonstrates how a method in a java class is supposed to be used, so a good indication that a java class might need a redesign is that the unit test is difficult to write. The 'code smells' that Martin Fowler refers to in his Refactoring book apply, but others indicators in unit tests that a certain class may require refactoring include:

- Unit tests must assert many conditions before calling the method they are testing.
- Unit test setup and teardown is a long and tedious process.
- A large number of assertions are required for each unit test and cannot be shared amongst other unit tests that might be testing the same feature.
- Each unit test method is extremely long and there is no common way of sharing a setup or teardown for each method.
- Developers spend excessive amounts of time in comparison to the time spent on the typical test-code-test cycle.

5.2.15 Managing Test Data

Rigourous unit testing may result in the creation and maintenance of a large amount of test data. There are a number of best practices related to the managing of this test data, and each should be selected to suit each circumstance. There are two types of data, actual and expected. Actual test data is the data that is being tested against. Expected test data is the data that you expect to be returned as a result of some function. For clarity, if we are testing method, f, this can be written as f(actual) =expected. That is, we test the expected result of the function based on input actual values. The techniques for managing test data can be classified into two main categories. These are:

1. Managing Actual Data
    a. **Unit Test Managed Data** - SQL scripts that forcefully inserts test data into the database is a useful mechanism for ensuring that the environment assumed for a unit test is maintained. Data assumed to be inserted correctly should be documented in unit tests and should be customised so that it is setup/teardown at the best possible points (e.g. one time setup per class, or one time setup per suite).
    b. **Permanent Data** - Similar to the previous method, this technique assumes that the data is in the database. It is the responsibility of the unit tester to validate its existence before using it, or reinserting it when it is not found.

c. **Java Class Data Factory** - This is the programmatic way of specifying unit test data and is one that java developers may feel more comfortable with. A data factory type of class applies the extract class refactoring technique to loosen the coupling between the unit tests and the creation of objects used by the unit tests. This technique is useful when a number of non-related unit tests depend on the same objects (and their states) for execution. Note that Java Class Data Factory can be used for managing either actual or expected test data.

2. Managing Expected Data

   a. **Text File Data** - An extension to JUnit, referred to as JUnit++, allows for configurable unit tests. The text file externalises the test data into a number of properties, allowing for highly configurable unit tests. This method is useful if you do not want to recompile unit tests due to a change in the test (actual) data. This method leaves the test data loosely coupled from the unit test, which may or may not be desirable depending on the needs of the unit test. Healthcare implemented a variation of JUnit++ by creating a ConfigurableTestCase class. It overcame some limitations of the JUnit++ extensions. For example, JUnit++ requires a configuration file whereas it is optional with ConfigurableTestCase. More information on the ConfigurableTestCase class (also known as the JUnitBL extension) can be found in the Appendix.

5.3 Running Unit Tests

5.3.1 Define standard Ant targets

There are many benefits to using Ant as a tool for managing the tasks involved with your java project, the most relevant here is its portability. Common targets across the system make the process of running unit tests much easier. Class path properties can be shared system wide without changes being required. A good build file will also assist developers in setting up their own individual targets that might execute a certain subset of unit tests that they might be working on.

A standard ant target should also be defined to execute all unit tests for the entire project. This allows all developers to run the testing environment in the same way and makes way for the next best-practice.

5.3.2 Execute all unit tests continuously

Running all the unit tests for a system 24 hours a day ensures that the system is constantly in a valid state. Changes to the code base or changes to the environment that may cause a unit test to fail will be picked up if tests are run all day continuously. It is not really that feasible to expect a software engineer or even a test engineer to run the tests all the time, it is better to automate this process. Now generally accepted as 'Continuous Integration'[16], there are a number of options that can be applied to get unit tests running all the time. These include:

- Running a hand-written daemon process that updates its own code base, compiles code and then invokes the main method on a particular class or set of classes.
- Cruise Control (http://cruisecontrol.sourceforge.net/) – An open source java program developed by the Thought Works© company that was devised to act as a continuous build and testing tool. Its Java based and object-oriented design are geared to providing a generic framework to operate with various source control mechanisms. Execution of unit testing can be incorporated as part of its build process.

[16] Martin Fowler's Article 'Continuous Integration' - http://www.martinfowler.com/articles/continuousIntegration.html

- Ant Hill (http://www.urbancode.com/projects/anthill/) – A build management server that ensures a controlled build process and promotes the sharing of knowledge within an organisation. It allows for unit testing to be integrated with its cycle and has the ability to publish artefacts upon completion.

Regardless of the tool used to automate the process, the continuous running and reporting of the entire unit test suite for a project can be useful in monitoring the state of that piece of software.

5.3.3 Code Coverage

The vigilant developer or tester will ensure that there are plenty of unit tests for their code, writing test plans and validating that tests exist for all conditions. As a system grows however, manually validating that the test plans cover all possible branches of code becomes far too difficult. Code coverage refers to the amount of code that is actually tested with the unit test suite.

There are a number of tools available for reporting how well the unit test suite covers the code it is testing. Some of these include:

- Clover (http://www.thecortex.net/clover) - A commercial tool that integrates with JUnit and Ant to provide a number of reports for JUnit unit test coverage.
- Rational Purecoverage (http://www.rational.com) - A commercial tool developed by the Rational Group from IBM that comes in a Windows and a Unix version. This tool can check coverage for a number of languages including VB, VB.NET, C, C++, C# and Java.
- TCAT (http://www.soft.com/Products/index.html) - Offered by Software Research, Inc, TCAT is a part of their comprehensive suite of software test tools, that provides coverage reports for languages including C, C++ and Java.
- Glass Jar Toolkit (http://www.testersedge.com/glass.htm) - A commercial coverage tool for testing java programs.
- OptimizeIt! (http://www.borland.com/optimizeit) - Typically used as a profiling tool for testing java application's memory usage, this tool also includes a feature for analysing code coverage.
- Quilt (http://quilt.sourceforge.net/) - An opensource project, still in an alpha stage that aims to provide a report of code coverage when integrated with JUnit unit tests.

Figure 8 depicts a package overview report that was generated by Clover for the open source CheckStyle project (http://checkstyle.sourceforge.net). The report here describes the percentage code coverage for each class within a particular package (on the left most panel). The right most panel contains a summary for the packages that Clover was run for.

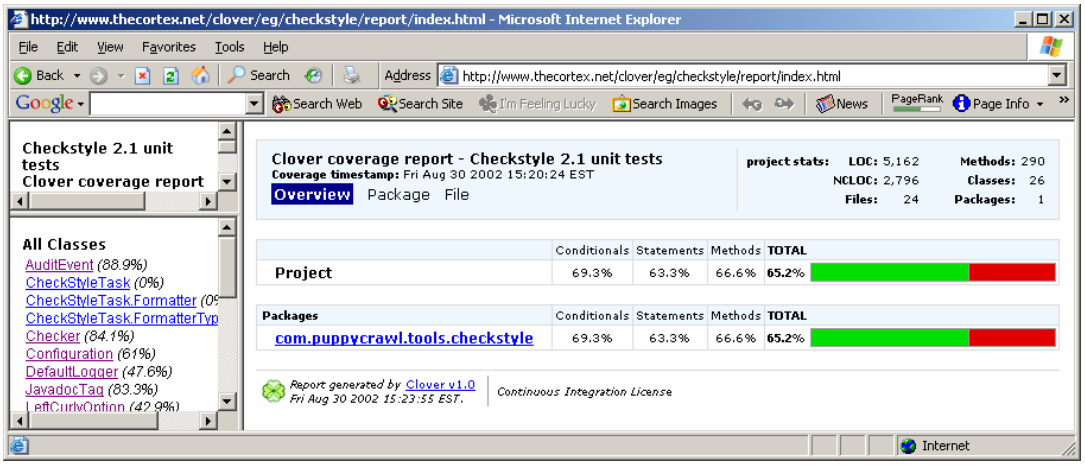

Figure 8: Clover Package Overview report

Selecting a class on the left hand pane results in the class being displayed on the right most panel like in Figure 9. This detailed summary is a copy of the source file that the coverage report was run on. Each significant line in the source file has two numbers. The first represents the line number in the file, the second is the number of times that line was executed. Lines that were not executed are highlighted in a pink colour, indicating the need for additional unit tests to be written for this block of code (assuming its complexity makes it worthwhile testing).

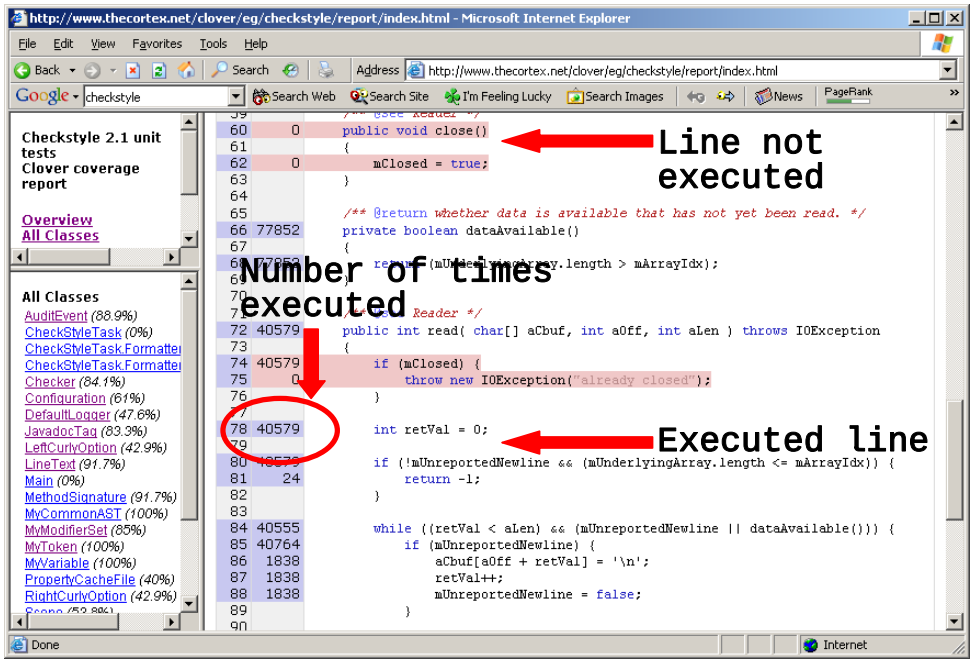

Figure 9: Clover Class Coverage Report

6 CONCLUSION

Testing remains an essential part of any software engineering process. Unit testing is one of the lowest forms of testing, aimed at testing the individual components that an application may be built with. This type of testing has been carried out historically ad hoc, with no formal way of way of writing and managing unit tests. As a result, it has been typically neglected and the value it can add by improving software quality and reducing future maintences costs are never realised.

With the acceptance of JUnit, the java-based unit testing framework as an industry de facto standard for writing unit tests, any java-based software project can easily integrate unit testing into their development environment. JUnit's maturity as an open source project allow projects to leverage industry experience and extensions to the framework all at a zero licensing cost.

JUnit is not the answer to all testing requirements, lacking the ability to easily test things such as GUIs and EJB classes. It does not provide a complex reporting mechanism and sometimes it can time to customise the framework for individual applications. There are a number of extensions or other testing products that can be used to fulfill those requirements that JUnit cannot.

JUnit it centred around two main classes that both implement the Test interface. The TestCase class is used as the container for defining unit tests, the other, the TestSuite class is used as a way of managing those tests. There are also a number of classes that the framework provides for running unit tests that report the results in a number of ways such as in textual or graphical format. This document has outlined a number of best practices for setting up the test environment, writing the unit tests and running the tests. It has also briefly discussed how BC4J components have been tested.

Unit testing is a great skill for any software engineer to develop as it can help in writing less bug-free code. However it is important to keep in mind that having unit testing does not guarantee the quality of the code. Thousands of poor quality unit tests will give a false impression of the quality of a piece of software. Writing high quality unit tests will help in some way, but you must make sure that the code you are testing is just as good.

7 APPENDIX A: EXAMPLE CODE

7.1 HitCounter.java

```
package oracle.apps.example;

import java.io.Serializable;
import java.util.Date;

/**
 * This class is a simple test class that is intended to demonstrate how to
 * construct a unit test to test the methods contained in this class.
 *
 * An instance of this class represents a count of the number of hits a web
 * page may have. Each page may or may not have its own hit counter. A hit
 * counter stores an internal number of hits that have been recorded, and
 * the date at which the hit counter was first created.
 */
public class HitCounter implements Serializable
{
  private int numberOfHits;
  private Date creationDate;

  /**
   * Construct a new counter that does not have any hits as yet.
   */
  public HitCounter()
  {
    creationDate = new Date();
    numberOfHits = 0;
  }

  /**
   * Returns the number of hits that this HitCounter has currently
   * observed.
   *
   * @return The number of hits that this Hitcounter has observed.
   */
  public int getNumberOfHits()
  {
    return numberOfHits;
  }

  /**
   * Returns the date that this HitCounter was created.
   *
   * @return The date that this HitCounter was created.
   */
  public Date getCreationDate()
  {
    return creationDate;
  }

  /**
   * Triggers an increment in the number of hits held by this HitCounter.
   */
  public void incrementNumberOfHits()
  {
    numberOfHits ++;
  }

  /**
   * Generate a unique hash integer for this HitCounter.
   *
   * @return a number that is a unique hash code for this HitCounter.
   */
  public int hashCode()
  {
    int result;
    result = numberOfHits;
    result = 29 * result + creationDate.hashCode();
    return result;
  }
```

```
  /**
   * Implementation of the equals method for a HitCounter. A
   * hit counter can be considered the same if and only if the
   * number of hits and the creation date of the hit counters
   * are the same.
   *
   * @return <code>true</code> if the object is considered the same as this
   *         HitCounter, otherwise <code>false</code>
   */
  public boolean equals(Object object)
  {
    if ( object == null ) { return false; }
    if ( this == object ) { return true; }
    if ( !(object instanceof HitCounter) ) { return false; }

    final HitCounter hitCounterToCompare = (HitCounter) object;
    if (numberOfHits != hitCounterToCompare.numberOfHits) { return false; }
    if (!creationDate.equals(hitCounterToCompare.creationDate))
    {
      return false;
    }

    return true;
  }

  /**
   * Implementation of the standard toString() method, resulting
   * in a string that describes the current values of this instance
   * of the HitCounter.
   *
   * @return A string that represents a description of the current
   *         values of this HitCounter.
   */
  public String toString()
  {
    return "HitCounter{" +
           "numberOfHits=" + numberOfHits +
           ", creationDate=" + creationDate +
           "}";
  }
}
```

7.2 HitCounterTest.java

```
package oracle.apps.example;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.textui.TestRunner;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Date;

/**
 * This class was designed to test all the methods that were added or modified
 * in the oracle.apps.example.HitCounterTest class.
 *
 * Note that not this is not yet a comprehensive test suite for this methods
 * contained in the HitCounter class.
 */
public class HitCounterTest extends TestCase
{
  private HitCounter hitCounter;

  public HitCounterTest(String testCaseName)
  {
    super(testCaseName);
  }

  public static Test suite()
  {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(HitCounterTest.class);
    return suite;
  }

  public static void main(String[] args)
  {
    TestRunner.run(suite());
  }
```

```
protected void setUp()
{
  hitCounter = new HitCounter();
}

/**
 * Tests that the increment number of hits method on the HitCounter
 * class is operating as expected.
 *
 * When we effectively 'hit' the HitCounter, we should expect to
 * increment exactly the number of times we hit it.
 */
public void testIncrementMethod()
{
   final int CURRENT_NUM_OF_HITS = hitCounter.getNumberOfHits();
   final int NUMBER_OF_HITS = 10;
   for ( int i = 0; i < NUMBER_OF_HITS; i++ )
   {
     hitCounter.incrementNumberOfHits();
   }
   assertEquals("Number of expected hits was different",
                CURRENT_NUM_OF_HITS + NUMBER_OF_HITS,
                hitCounter.getNumberOfHits() );
}

/**
 * This validates that the toString() method is operating correctly.
 * toString() should produce a useful debugging string that details
 * the values of its fields.
 */
public void testToString()
{
  final int CURRENT_NUM_OF_HITS = hitCounter.getNumberOfHits();
  final Date CREATION_DATE = hitCounter.getCreationDate();
  final String EXPECTED_STRING = "HitCounter{numberOfHits=" +
                                 CURRENT_NUM_OF_HITS + ", creationDate=" +
                                 CREATION_DATE + "}";
  assertEquals("toString() method does not produce the expected output",
               EXPECTED_STRING, hitCounter.toString() );
}

/**
 * Test that a null passed into the equals() method returns false.
 */
public void testEqualsWithNullValue()
{
   assertTrue("equals(null) does not result in a false boolean value",
              !hitCounter.equals(null) ); // note the ! symbol
}

/**
 * Test that validates that a call to the equals(Object) method with
 * the same reference results in true.
 */
public void testEqualsWithSameReference()
{
  assertTrue("equals(this) does not result in a true boolean value",
             hitCounter.equals(hitCounter) );
}

/**
 * Create an object that captures exactly the same data as the
 * original hit counter, but are different references. Make sure
 * that the equals() method returns true for the comparison
 */
public void testEqualsWithPerfectClone() throws Exception
{
  HitCounter copyOfHitCounter =
    (HitCounter)cloneViaSerialization(hitCounter);
  assertTrue("equals() method of hitcounter should return true with a " +
             "copy of a hitcounter",
             hitCounter.equals(copyOfHitCounter));
  assertTrue("equals() method should work in reverse",
             copyOfHitCounter.equals(hitCounter));
}

/**
 * Validate that the hash code for a hit counter and its perfect
 * clone is the same.
 */
public void testHashCodeWithPerfectClone() throws Exception
{
  HitCounter copyOfHitCounter =
     (HitCounter)cloneViaSerialization(hitCounter);
  assertEquals("Hashcode for the hit counter and its perfect clone " +
               "should be the same",
               hitCounter.hashCode(), copyOfHitCounter.hashCode() );
}
```

```
  /**
   * Validate that the hash code is generated consistently regardless
   * of the time that passes.
   */
  public void testHashCodeIsConsistent() throws Exception
  {
    assertEquals("HashCode generated for the same object should be " +
                 "consistent",
                 hitCounter.hashCode(), hitCounter.hashCode() );
  }

  /**
   * Clones an object via serialization. The object will be written to an
   * in memory output stream, and read back in from that stream and
   * reconstructed into the object that it should have been.
   *
   * @param serializableObject An object that is serializable
   * @return A copy of the object where all serializable fields have the
   *         same value as that of the original.
   * @throws IOException If there is a problem writing the object
   * @throws ClassNotFoundException If the object passed in does not have
   *         its class definition loaded into memory.
   */
  protected Object cloneViaSerialization( Object serializableObject )
    throws IOException, ClassNotFoundException
  {
    ByteArrayOutputStream inMemoryOutputStream =
      new ByteArrayOutputStream();
    ObjectOutputStream serializer =
      new ObjectOutputStream(inMemoryOutputStream);
    serializer.writeObject(hitCounter);
    serializer.flush();

    ByteArrayInputStream inMemoryInputStream =
      new ByteArrayInputStream(inMemoryOutputStream.toByteArray() );
    ObjectInputStream deserializer =
      new ObjectInputStream(inMemoryInputStream);
    return deserializer.readObject();
  }

}
```

8 APPENDIX B: ONE TIME SETUP

Sometimes it is necessary to improve the speed with which tests are run. When testing in an environment where there is a large amount of set up and teardown, it might be easier to perform a one-time setup of test data for an entire **test suite** instead of for each **test**.

Consider the following block of code. Although the test methods could be pushed into a single test method, each has been separated to demonstrate the one-time setup principle.

```
package oracle.apps.example;

import junit.framework.TestCase;
import junit.framework.Test;
import junit.framework.TestSuite;

import java.util.Set;

/**
 * This class validates that the titles (Mr, Mrs, Ms, and Miss) are always
 * valid for the lookup scheme 'TITLE_LOOKUP'.
 */
public class TitleLookupTest extends TestCase
{
  /**
   * The name of the lookup scheme for titles.
   */
  public static final String TITLE_SCHEME_NAME = "TITLE_LOOKUP";

  /**
   * A class that lets us execute a set of defined SQL scripts. The script
   * must exist relative to the base classpath.
   */
  protected static SQLScriptRunner sqlScriptRunner = new SQLScriptRunner();

  /**
   * The resulting lookup set.
   */
  protected Set titleLookupSet;


  public TitleLookupTest ( String name )
  {
    super(name);
  }

  /**
   * Set up the lookup values in the database
   */
  protected void setUp() throws Exception
  {
    sqlScriptRunner.executeScript("/lookups/insert_title_lookups.sql");

    // This finds all of the lookups for each test
    LookupService lookupService = LookupService.getService();
    titleLookupSet = lookupService.findLookupSet(TITLE_SCHEME_NAME);
  }

  /**
   * Remove the lookup values in the database
   */
  protected void tearDown() throws Exception
  {
    sqlScriptRunner.executeScript("/lookups/delete_title_lookups.sql");
  }

  /**
   * Tests that the title 'Mr' is a valid title for the title lookup scheme.
   */
  public void testValidTitleOfMr()
  {
    assertTrue("The title lookup scheme does not contain the valid " +
               "title 'Mr'", titleLookupSet.contains("Mr") );
  }

  /**
   * Tests that the title 'Ms' is a valid title for the title lookup scheme.
   */
  public void testValidTitleOfMs()
  {
    assertTrue("The title lookup scheme does not contain the valid " +
               "title 'Ms'", titleLookupSet.contains("Ms") );

  }
```

```
  /**
   * Tests that the title ‘Miss’ is a valid title for the title lookup
   * scheme.
   */
  public void testValidTitleOfMiss()
  {
    assertTrue("The title lookup scheme does not contain the valid " +
               "title ‘Miss’", titleLookupSet.contains("Miss") );
  }

  /**
   * Tests that the title ‘Mrs’ is a valid title for the title lookup
   * scheme.
   */
  public void testValidTitleOfMrs()
  {
    assertTrue("The title lookup scheme does not contain the valid " +
               "title ‘Mrs’", titleLookupSet.contains("Mrs") );

  }

  /**
   * Validate that the title of ‘Dr’ is not considered a valid title for
   * the title lookup scheme.
   */
  public void testInvalidTitleOfDr()
  {
    assertTrue("The title lookup scheme should not consider ‘Dr’ as a " +
               "valid title", !titleLookupSet.contains("Dr") );
  }

  /**
   * Validate that the number of titles in the title lookup scheme only
   * contains 4 values.
   */
  public void testNumberOfTitles()
  {
    assertEquals("Number of titles in the title scheme was incorrect",
                 4, titleLookupSet.size() );
  }

  /**
   * Return a test suite containing all tests for this class.
   * @return a test suite containing all tests for this class.
   */
  public static Test suite()
  {
    return new TestSuite(TitleLookupTest.class);
  }
}
```

The code listed above initialises the lookup seed data before every test and restores the database to its original state after each unit test. This means that the SQL scripts are executed before and after **every** test. This is only a simple example, but demonstrates that sometimes it would be more useful to perform a one time set up for the entire test suite instead of once for every test to improve the performance.

To do this, we leverage the TestSetup class.

```
...
  /**
   * The SQL script has been pushed into the wrapper class for a one-time
   * set up. We still set up variables that we need for unit tests here.
   */
  protected void setUp()
  {

    // This finds all of the lookups for each test
    LookupService lookupService = LookupService.getService();
    titleLookupSet = lookupService.findLookupSet(TITLE_SCHEME_NAME);
  }

  /**
   * The SQL script has been pushed into the wrapper class for a one-time
   * set up.
   */
  protected void tearDown()
  {
     // Nothing to do here anymore
  }
```

```
  /**
   * We wrap up the test suite in an anonymous extension to the TestSetup
   * class. This lets us locally define a setUp() and tearDown() that is
   * executed before the tests in the suite are executed.
   */
  public static Test suite()
  {
    TestSuite suite = new TestSuite(TitleLookupTest.class);
    TestSetup wrapperSuite = new TestSetup( suite )
    {
      protected void setUp()
      {
        sqlScriptRunner.executeScript(“/lookups/insert_title_lookups.sql”);
      }

      protected void tearDown()
      {
        sqlScriptRunner.executeScript(“/lookups/delete_title_lookups.sql”);
      }
    };
    return wrapperSuite;
  }

...
```

As shown in the code above, performing a one-time set up and teardown can help to drastically improve the performance of some unit tests. This princple can be extended to apply to any point in a test hierarchy, making it easy to help reduce the cost of common code being run for every single unit test.

9 DOCUMENT REFERENCES

JUnit A Cook's Tour

Webpage: http://junit.sourceforge.net/doc/cookstour/cookstour.htm

Written by: Erich Gamma and Kent Beck

Description: A description of the way in which JUnit is designed written by the original authors Erich Gamma and Kent Beck.

JUnit Best Practices

Webpage: http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html

Written by: Andy Schneider from JavaWorld

Description: A listing of best practices useful for developers getting started and those who want to refine their unit testing process.

JUnit Primer

Webpage: http://www.clarkware.com/articles/JUnitPrimer.html

Written by: Mike Clark

Description: A useful introduction to the JUnit framework. As mentioned on the webpage, 'This article demonstrates how to write and run simple test cases and test suites using the JUnit testing framework.'

JUnit

Webpage: http://www.junit.org

Description: The official home page for the website that hosts all the JUnit documentation.

Continuous Integration

Webpage: http://www.martinfowler.com/articles/continuousIntegration.html

Author: Martin Fowler

Description: An article that describes the benefits derived from continuous integration of code changes and unit tests.

Managing the Development of Large Software Systems

Copyright: IEEE 1970

Author: Dr Winston W Royce

Description: First published from Proceedings, IEEE WESCON, August 1970, Royce discusses his experiences and observations from a number of software projects to propose a model of describing the lifecycle of projects.