Getting Started With Unit-testing

An Oracle White Paper
December 2004

ORACLE

INTRODUCTION 3
WHAT IS UNIT-TESTING? 3
TEST-FIRST PRINCIPLE 4
USING A TEST FRAMEWORK 6
AUTOMATED TESTING 6
UNIT TESTING FRAMEWORKS 7
  JUnit 7
  Mock Objects 7
  Apache Cactus 8
  HttpUnit / ServletUnit 8
  utPLSQL 8
  Clover 8
  Other Frameworks 10
BEST PRACTICES 11
  Using Test Data 11
  SQL insert scripts 11
  Java Classes 12
  Export files 12
OTHER BEST PRACTICES 12
  CRUD Test Persisting Classes 13
  Follow Class Hierarchy 13
  Use Test Fixtures 13
MORE INFORMATION 14
CODE EXAMPLES 14
  Example Code Class: StringUtils 14
  Example Code Class: Person 15
  Example Test Case: StringUtilsTester 17
  Example Test Case: PersonTester 18
  Example Test Suite 19
  Example Test Fixture 20
  Example Unit Testing with ADF BC 21
    ADF Test Fixture 21
    ADF CRUD Test 22

INTRODUCTION

Do you as project manager have the final responsibility of the quality of code being produced, but to be honest you do not really know what that quality is? Are you strongly advised by your developers that code should be refactored, but you must say no to avoid issues with already finished code? Are you ever afraid of specific developers leaving the project, because they are the only ones that know how important pieces of code work? Read on and find out how unit-testing can give you insight in the real progress of your project and prevent bottlenecks in your developers staff.

Do you as a developer sometimes feel an urgent need to refactor existing code but are you afraid to touch it, too scared to break it? Do you now and then get sick of being bothered with issues regarding code you hoped never to touch again? Or do you simply want to begin with unit-testing but you don't have any idea where to start? Read on and learn how you can improve the quality of your code and make sure that changes you make won't break it.

WHAT IS UNIT-TESTING?

A unit test is a test of a distinct unit of functionality. A unit of functionality is not directly dependent on the completion of another unit of functionality.
Within a Java application a unit of functionality is often (but not always) a single method. Unit tests are the finest level of granularity in testing.

Setting up a unit test can take considerable time, sometimes even more than the creation of the unit of functionality. To justify this effort, unit tests should add to the quality of the code and be reusable. In the end unit testing should pay by delivering more functionality in the same time or the same functionality in less time. Most of this is achieved by that unit-testing reduces the effort needed for integration and acceptance testing, because the majority of design and programming errors that are made are discovered early in the development cycle. Learning from that, this prevents developers from making the same errors over and over again. Ideally during acceptance testing one can suffice with verifying that the system meets the specifications without being hindered, let alone frustrated, by coding errors.

In practice, this is very hard if not impossible to prove for one specific project. There are however strong indications that an effective way of unit testing does just that. An effective way of unit testing thereby consists of:

- Applying the test-first principle,
- Using a test framework and
- Automating the tests.

In the following three sections will be explained how this adds to effective unit-testing.

TEST-FIRST PRINCIPLE

The test-first principle means that the test is written before the actual code. In practice you test a little, code a little and so on. This could go like this:

First you write a test that calls a method that does not yet exist. Why would you do a silly thing like that? Well, when writing the test you need to think about the parameters you need to provide and what the result is that the method should return.

Suppose you want to write a substitute() method that substitutes one String by another. The test for this method could look like this:

```
  public static void testSubstitute()
  {
    assertEquals("a bear", StringUtils.substitute("a goose",
"goose", "bear"));
  }
```

Compilation of the test will fail of course, as the substitute() method does not yet exist. So the next step is to write the method with the appropriate signature and let it return a value that makes the test run. Initially you can return a hard-coded value:

```
  public static String substitute(String text, String replace,
String by)
  {
    return new String("a donkey");
  }
```

When you try to run this test it will fail, as you expect it to return 'a bear', not 'a donkey'. It is always good to let the test fail at least once, to validate it actually works.
Now you change the test to return a value that will make the test run successfully:

```
  public static String substitute(String text, String replace,
String with)
  {
    return new String("a bear");
  }
```

From now on, you can verify with each step you take that you did not break the code of the substitute() method!

The next step would be to replace the hard-coded return value by some code that actually replaces all occurrences of the 'replace' parameter by the 'with' parameter. You gradually add extra tests like one to assert that not only the first but *all* occurrences are replaced and to assert that null parameters are dealt with properly. Each test case you add you run immediately. When it fails you fix the code until it runs successfully, and so on.

Some people distinguish between *test-first* and *test-driven development* (where other people say it's one and the same), test-first being the 'extreme' form of test-driven development. Test-driven development refers to writing code together with the test. Anyway, whatever you call it, either way you are likely to find that it influences the way you write your code. If you were not doing that already, you probably find that you will create smaller methods with clearer responsibilities and a cleaner interface. Whenever necessary, you might even write code only to be able to unit-test it.

As will be discussed later on, IDE plug-ins that support unit-testing create stubs for tests based on existing methods. This does not encourage test-first in its most extreme form. No matter how you do it, remember and practice the idea behind the test-first principle, being that you should create a test as soon as possible and not after you finished the method. Obviously you would miss the most important advantage of unit-testing which is helping you to discover errors as soon as you make them.

The advantage of applying the Test-First principle is threefold:

- First it encourages you to start with thinking about *what* a unit of functionality should accomplish, instead of *how* it should be done. This helps in preventing you from wasting time on writing code that is not necessary for this functionality.
- Secondly it provides you a mechanism to test each step you take in writing the code, enabling you to discover errors as soon as they occur.
- Thirdly it will guide you in writing clearer methods with cleaner interfaces and responsibilities.
- In this way applying the Test-First principle helps you to write better code in less time.

It should be clear by now that unit-testing should be done by the developers themselves, rather than by a test team that creates unit tests afterwards. The fact that errors are discovered when and where they arise together with better-constructed code, outweighs the advantage of detecting blind spots by a test team. To ensure that unit-testing is done consistently and of good quality, you can either let the developers peer review each other, or let a unit-testing coordinator perform reviews. Furthermore, mind that unit-testing should be succeeded by integration testing, which also serves as a 'safety net' for unit-tests of bad quality.

Depending on their knowledge, you could also consider involving the customer in unit-testing and letting them determine the test cases for you, or help you to determine them.

USING A TEST FRAMEWORK

Looking at the previous example, you will notice that an assertEquals() method has been used. Apparently this method verifies, or *asserts*, that the actual result of the statement 'substitute("a goose", "goose", "bear")' equals the expected result *'a bear'*. It probably will give a neat error message when the assertion fails. But not all tests can be implemented by comparing two Strings with each other. Basically we would like to be able to compare an object of any type with some other object of the same type.

Also, most applications will consist of many methods for which even more test cases should be written. To verify that changes in the code have not broken the application, all tests must be run on a regular base. Rather than needing to run the tests one by one, one would like to be able to run sets of tests together. When one assertion fails, other tests should continue. In the end, clear feedback must be provided about what assertions failed, if any.
This is where a unit-testing framework comes in place. A good testing framework offers at least the following functionality:

- Convenience methods to do assertions based on many object types.
- The ability to run any test any time, to support bug fixing and refactoring.
- The ability to create 'test suites' that consist of sets of tests that can be run together.
- A user-friendly interface that gives an overview of all executed tests. When one ore more tests failed, it will indicate them.

The only practical way of doing unit-testing for an application of any significant size, is by using a unit-testing framework.

AUTOMATED TESTING

For applications of a significant size, one soon will feel the need to automate the unit tests. This means that all tests are run automatically on a regular base and the results are logged to be reviewed afterwards. The developers can than restrict themselves to running unit tests for the code they created or modified and all other code that might be affected by this directly. At regular intervals it is automatically verified if all test run successfully. If not, the first action is to fix the code or the tests (whatever is causing the problem). In this way the automated unit tests provide a kind of safety net, in that it helps to discover unanticipated side effects of new or changed code and to detect bugs at an early stage.

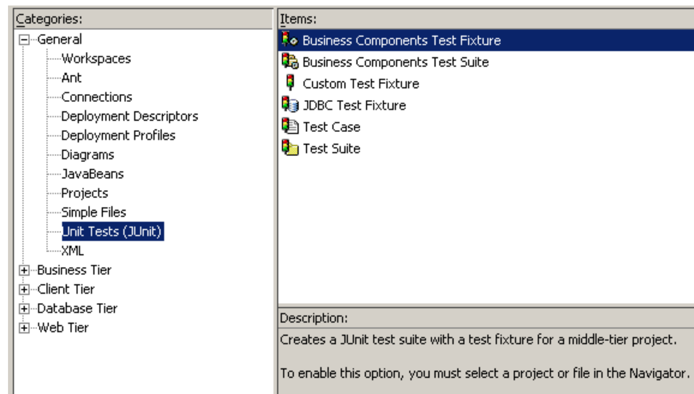
In it's most simple form, unit-testing is automated by creating a hierarchy of test suites, with one or only a few test suites at the top that can be run automatically and that run all unit tests.

Automate the tests. Run all tests during a nightly build and start the next day with fixing the tests that failed.

UNIT TESTING FRAMEWORKS

JUnit

The de facto unit-testing framework for Java is JUnit (open source). When you have an account on OTN, you can plug JUnit easily into JDeveloper from the menu: Help -> Check for Updates -> check the JUnit Extension -> next. It will automatically download the JUnit plug-in files to your [JDeveloper home]\jdev\lib\ext directory (in Oracle JDeveloper 10.1.2 and previous versions). You can also download a stand-alone version from http://www.junit.org. JUnit offers all the features mentioned before, and can test Java classes that run in a client JVM. Using the plug-in you can easily create unit test stubs for existing methods and create test suites for a set of existing unit tests (see figure below). You can also create a set of reusable test fixtures, which will be discussed later. When you want to follow the test-first principle, you should create a test stub for a method right after you defined its signature, before you write the actual code. In the code examples at the end of this document a detailed example is presented of the usage of JUnit.



JDeveloper JUnit plug-in

Mock Objects

There is a lot of code that depends on the existence of other software components, for which making a unit test results in a complex setup. Think about persistence code that needs to connect to a database, or code that simply will not run in a client JVM, like servlets that process HTTP requests and responses, or remote interfaces of EJB that need to connect to a remote server. To a certain extend you can prevent the need to run in a broader context by making use of so-called Mock Objects. For example, instead of setting up a database connection and retrieve objects from the database, you could use mock objects that behave like database objects. There is a specific framework for developing and using mock objects that you can download from http://www.mockobjects.com (open source).

Apache Cactus

Whenever mock objects are not an option, and you need to integrate with other software components, you start using Apache Cactus. Cactus extends JUnit and enables unit testing from a container like OC4J or Tomcat. In this way Cactus enables testing of anything that runs within a container, like EJB's, servlets, taglibs, servlet filters, JSP pages and XSLT. You can download Cactus from http://jakarta.apache.org/cactus/ (open source). As you can see on their web-page, the creators of Cactus claim that it makes not only that you do not need to create mock objects, but you can also do functional testing where you otherwise would need to use a framework like HttpUnit, which is the next framework to be discussed.

HttpUnit / ServletUnit

With HttpUnit you can emulate the HTTP requests of a web application. HttpUnit emulates (parts of) the behavior of web browsers, including form submission, JavaScript, basic form authentication, cookies, and page redirection. Together with ServletUnit (that emulates a servlet container) and JUnit, HttpUnit is an alternative for using Cactus, as long as you do not necessarily need to test from within a container. HttpUnit and ServletUnit can both be downloaded from http://httpunit.sourceforge.net/ (open source). Mind that, as HttpUnit does not emulate a browser, you cannot do UI testing with it, like performing validations or other actions within the browser.

utPLSQL

When using the Oracle database, there often will be code that is developed using PL/SQL, like background batch processes. PL/SQL can be unit-tested using the utPLSQL that can be downloaded from http://utplsql.sourceforge.net/ (open source).

Clover

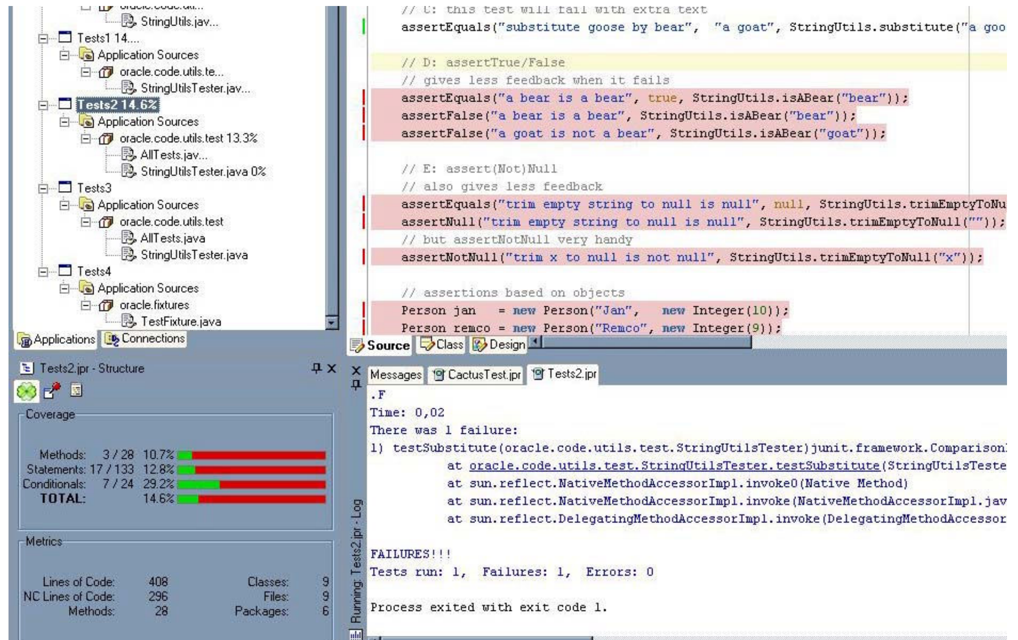
Another important aspect of unit-testing is monitoring the coverage, referring to what code actually is hit by a unit test. This can be done using Clover. You can integrate Clover into the build process using Jakarta Ant build files. Clover will add specific code to the class files to log what is being executed. Using this log, it can report what methods are executed and within a method what statements, how often and in what order. In this way it can even help you to restructure the code to run more efficiently.
You can also integrate Clover into JDeveloper 10g, allowing you to view what was executed visually in the code itself (see figures below). You can download an evaluation copy of Clover from http://www.thecortex.net/clover/. Clover is not for free, but highly recommendable on an average to large project.

```
20        */
21 1     public UserContext(String userName, boolean hnhFlag, String huidigeRegistratie) {
22 1         this.userName = userName;
23 1         this.hnhFlag = hnhFlag;
24 1         this.huidigeRegistratie = huidigeRegistratie;
25       }
26
27       /**
28        * Method UserContext.
29        *
30        * Context om de parameters voor           te zetten
31        *
32        * @param userName Initiale parameter voor het zetten van de username.
33        */
34 0     public UserContext(String userName) {
35 0         this(userName,false,"");
36       }
37
38       /**
39        * @see
40        */
41 1     public String getUsername() {
42 1         return this.userName;
43       }
44
45       /* (non-Javadoc)
46        * @see
47        */
48 0     public boolean getHitNoHitFlag() {
49 0         return this.hnhFlag;
50       }
51       /* (non-Javadoc)
52        * @see
53        */
54 0     public String getHuidigeRegistratie() {
55 0         return this.huidigeRegistratie;
56       }
57   }
```

Report generated by Clover v1.2.3
di mei 18 2004 14:11:35 CEST.

30 day Evaluation Version distributed via the Maven Jar Repository. Clover is not free. Yo

Example of a Clover Report



Clover can be integrated in JDeveloper

Anthill Pro

Anthill Pro is a software build management server. Using Anthill Pro you can schedule the build process (including the unit tests and Clover) based on builder frameworks like Apache Ant. As it integrates with version control systems like CVS, it can extract the source code from the version control system before it starts the build. In this way you can for example fully automate your nightly build. When it finishes it can send an email to the test manager and notify about the status of the build. It creates reports about the build process, especially letting you view what builds went wrong and what tests did fail. You can download an evaluation copy of Anthill Pro from http://www.urbancode.com/products/anthillpro. Anthill is not for free, but highly recommendable on an average to large project.

Other Frameworks

There are many more frameworks or extensions that leverage JUnit, most of them open source. Refer to the extensions page of JUnit.org (http://www.junit.org) to find out more. Among them frameworks or extensions for code coverage, performance testing, the build process, integration and acceptance testing, data loading, JSP testing and much more.

BEST PRACTICES

A few best practices have already been mentioned before. They are:

- Apply the test-first principle to maximize the effect of unit-testing
- Always use a unit testing framework
- Automate all unit tests to run on a regular base

The following best practices can be added to them.

Using Test Data

You should be able to run unit tests independent of each other and in every order. This implies that the test data that is used by one unit test should not be changed by any other unit test. In practice this can best be achieved by creating one 'base set' of test data. All unit tests can rely on this base set to be present. When a unit test needs to change data, it should create it itself. To prevent interference with any other unit test, it should delete the data it created afterwards.

The base set can consist of SQL insert scripts, of Java classes that create data to do this job or an export file that is being imported before running unit tests.

SQL insert scripts

Using SQL insert scripts has the following two advantages:

- They can be maintained by people that have no knowledge of Java.
- SQL insert script allow for testing database persistence logic that cannot be tested through the Java layer.

An example of the second bullet is inheritance in TopLink. When multiple classes are mapped on one table, you create a discriminator column that TopLink uses to determine the subclass of an object. You cannot create data through TopLink with values for the discriminator column other than the predefined ones. However, in SQL insert scripts you can insert invalid values, resulting in data that cannot be accessed through TopLink. To prevent this, you can create a check constraint that enforces the use of proper values for the discriminator column, or create a database trigger that does something similar.

Persistence logic is not (directly) related to business logic, but concerns logic needed for a proper working of the database or proper communication with the Java persistence layer. Most common examples are primary, unique and foreign key constraints.

For most applications of any significant size, is it very likely that data is going to be changed through other channels than the Java persistence layer. Examples are PL/SQL interfaces with other data sources, corrective maintenance of data, or data conversions during an upgrade of the system. If any of these situations might occur, you want to make sure that the persistence logic is tested.

When the SQL insert scripts are of a considerable size, the disadvantage is that maintenance becomes time consuming and error prone. So whenever this may become an issue, consider the other two alternatives.

Java Classes

Using data creating classes has the following advantages:

- It also tests the proper working of the persistence layer as far as inserting of data is concerned. This can help in discovering OO-relational mapping errors at an early stage.
- Code support provided by the IDE can help preventing errors. Using the Java compiler, the need for changes of the base set are discovered easily as far as removing or renaming of attributes or classes is concerned. In this way the base set is easier to maintain than in case of the other two options.

The disadvantage is that it will take more time to create the initial setup of data creating classes. However, you could consider to partially generate the initial setup.

Export files

Using export files has the following advantages:

- It offers support for testing data conversions as you can load a start situation and than apply the conversion scripts, which makes it especially a good alternative for systems that already have gone production.
- You can maintain the test data using utility tools like PL/SQL Developer or Toad.

The first advantage is also a disadvantage. The drawback is that you are also less flexible because *each* change of the structure of the database requires a data conversion, also the ones not relevant for system in production. You would like to prevent these, as conversion are often time consuming. Consider using export files as a sequel to using one of the other two alternatives after the system has gone into production, or for testing conversions.

OTHER BEST PRACTICES

Separate Test Code

You should separate test code from application code. Among other reasons, one of the more important ones is that you should not deploy test code to production. It complicates the deployment process, jar files become bigger than necessary and you might even introduce security risks as test code might expose methods that should be hidden. To be able to test package protected methods (when necessary) you place test code in a separate project, but within the same package as the code to be tested.

Do Not Test Setters and Getters

Unless you wrote specific code on a getter or setter method for Java bean property, there is not much added value in spending time on writing unit tests for these methods. Writing unit tests for getters and setters will take considerable time, while the chance that you discover a problem that would not have been discovered otherwise is very small, as most getters and setters at some point in time will be called by other methods that are provided with a unit test.

In general the rule is to write a test that could reasonably break. If it cannot break on its own, it's too simple to break.

CRUD Test Persisting Classes

Unlike testing setters and getters, there is much added value in testing the creation, retrieval, update and delete of persisting code for example when you use TopLink. It will help you finding most OO-relational mapping errors, and for example errors with sequences. In case of a create you can restrict yourself to supply only those attributes that are mandatory and associations. In case of an update you can restrict yourself to one or two attributes. Often you will be able to generate most part of the CRUD tests.

Do Not White-box Test

Unit-testing should focus on what the class should do rather than how it is implemented. When white-box testing a class you would also test all private and protected methods, which implies that when you change the implementation there is a big chance you need to change the test also. In this way white-box testing would hinder effective refactoring, where you would like to be able to use an existing test to prove that you have not broken anything. Likewise it is better to test against an interface than against implementations of the interface.

Follow Class Hierarchy

When creating unit-tests for a class hierarchy, you should set up a parallel hierarchy for your unit-tests. This has as an advantage that when running the unit tests of a subclass, all tests of the superclass will automatically run as well but than in the context of the subclass. When the superclass is abstract with abstract methods, you can also create an abstract test class with abstract unit tests. The subclasses of this test class than need to implement the abstract unit tests of the superclass, preventing you from forgetting to implement these tests.

Use Test Fixtures

Unit-testing is not much different from writing other code, implying you should prevent code duplication. One way to do that is by creating so called *test fixtures* that consists of reusable code that you can use to setup a context prior to a unit test and destroy it afterwards. This can be used for example to setup a database connection, instantiate some object with specific attribute values and so on.

MORE INFORMATION

More information about unit-testing in general and JUnit in particular can be found at the JUnit site (http://www.junit.org). You can download the Javadoc of JUnit from there, which includes an excellent FAQ. Other useful information about unit-testing can be found at the Wiki site (http://c2.com/cgi/wiki?UnitTest). A very comprehensive example of various features of ADF (including unit testing) can be found in the ADF Toy Store Demo (http://www.oracle.com/technology/products/jdev/collateral/papers/10g/adftoystore.html).

CODE EXAMPLES

Example Code Class: StringUtils

The following class implements a method called substitute() that substitutes all occurrences of certain String within another String, a method called isABear() that returns true whenever a String equals "bear", and a method trimEmptyToNull() that will convert an empty String "" to a null value.

```
package oracle.code.utils;

import java.util.List;
import java.util.ArrayList;
import java.util.StringTokenizer;

public class StringUtils
{
  private static String BEAR = new String("bear");
  private static String EMPTY = new String("");

  /**
   * Substitutes all occurrences of certain String within another
String
   *
   * Mind that this is just an example. Instead of this substitute
method
   * you normally would use in.ReplaceAll(find,newString)
   *
   * @param in: the string in which to replace
   * @param find: the (sub)string to replace
   * @param newString: the string to replace it with
   * @return String in which all occurrences of in have been
replaced by
   * newString
   */
  public static String substitute(String in, String find, String
newString)
  {
    // when either of the strings are null, return the original
string
    if ( in == null || find == null || newString == null)
    {
      return in;
    }

    char[] working = in.toCharArray();
    StringBuffer stringBuffer = new StringBuffer();

    // when the find string could not be found, return the original
string
    int startindex = in.indexOf(find);
    if (startindex < 0)
    {
      return in;
```

```
    }

    int currindex=0;
    while (startindex > -1)
    {
      for(int i = currindex; i < startindex; i++)
      {
        stringBuffer.append(working[i]);
      }
      currindex = startindex;
      stringBuffer.append(newString);
      currindex += find.length();
      startindex = in.indexOf(find,currindex);
    }

    for (int i = currindex; i < working.length; i++)
    {
      stringBuffer.append(working[i]);
    }

    return stringBuffer.toString();
  }

  /**
   * returns true when s equals "bear"
   *
   * @param s: the String to test
   * @return true when s equals "bear"
   */
  public static boolean isABear(String s)
  {
    return BEAR.equals(s);
  }

  /**
   * converts an empty String “” to a null value
   *
   * @param string: the String to convert
   * @return null in case of an empty String, string otherwise
   */
  public static String trimEmptyToNull (String string)
  {
    if (string == null || EMPTY.equals(string.trim()))
    {
      return null;
    }
    return string;
  }
}
```

Example Code Class: Person

The following class is a very simple implementation of a Person with a name and an age. It implements equal that needs to be unit-tested. Like explained earlier, we will not test getters and setters.

```
package oracle.code.model;

public class Person
{
  private String name;
  private Integer age;

  public Person()
  {
  }

  /**
   * non-default constructor
   *
   * @param name: the name of the person
```

```
     * @param age: the age of the person
     */
    public Person(String name, Integer age)
    {
      this.name = name;
      this.age = age;
    }

    /**
     * sets the name of the person
     *
     * @param name: the name of the person
     */
    public void setName(String name)
    {
      this.name = name;
    }

    /**
     * returns the name of the person
     *
     * @return
     */
    public String getName()
    {
      return this.name;
    }

    /**
     * sets the age of the person
     *
     * @param age: the age of the person
     */
    public void setAge(Integer age)
    {
      this.age = age;
    }

    /**
     * get the age of the person
     *
     * @return Integer age
     */
    public Integer getAge()
    {
      return this.age;
    }

    /**
     * compares a Person with this Person
     *
     * @param p: the person to compare with
     * @return true when this person and p are equal
     */
    public boolean equals(Object p)
    {
      // return true when p revers to this
      if (this == p)
      {
        return true;
      }
      // return false when p is not a Person
      if (!(p instanceof Person))
      {
        return false;
      }
      // return true when all attributes equal
      Person q = (Person)p;
      return (  (this.name == null ? q.name == null :
  this.name.equals(q.name))
             && (this.age == null ? q.age == null :
  this.age.equals(q.age))
             );
```

```
    }
  }
```

Example Test Case: StringUtilsTester

The following class contains the unit tests for the StringUtils class. The tests are explained in more detail inline. This class has been created with the JUnit plug-in of JDeveloper which created the testSubstitute() method stub without any code, as follows:

- Right-click the project that will contain the tests -> New -> General -> Unit Tests (JUnit) -> Test Case
- On the 'Subject' tab browse to the class for which the test case must be created. On the same tab a tree with methods appears.
- Expand the tree with methods and check the methods that should be included in the test case, in this case all.
- On the 'Class' tab name the class 'StringUtilsTester'. Accept the default Package Name ('oracle.code.utils.test') and Extends ('junit.framework.TestCase')
- Finish.

After that the names of the methods in the test case have been changed to apply to de facto standards for naming methods.

```
package oracle.code.utils.test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import oracle.code.utils.StringUtils;
import oracle.code.model.Person;

public class StringUtilsTester extends TestCase
{
  public StringUtilsTester(String sTestName)
  {
    super(sTestName);
  }

  /**
   * String substitute(String, String, String)
   */
  public void testSubstitute()
  {
    // A: this test will succeed
    assertEquals("a bear", StringUtils.substitute("a goose", "goose",
"bear"));

    // B: this test will fail
    assertEquals("a goat", StringUtils.substitute("a goose", "goose",
"bear"));

    // C: this test will fail with extra text
    assertEquals("substitute goose by bear",  "a goat",
                 StringUtils.substitute("a goose", "goose", "bear"));

    // D: assertTrue/False
    // gives less feedback when it fails
```

```
    assertEquals("a bear is a bear", true,
StringUtils.isABear("bear"));
    assertTrue("a bear is a bear", StringUtils.isABear("bear"));
    assertFalse("a goat is not a bear", StringUtils.isABear("goat"));
  }

  /**
   * String trimEmptyToNull(String)
   */
  public void testTrimEmptyToNull()
  {
    // E: assert(Not)Null
    // also gives less feedback
    assertEquals("trim empty string to null is null", null,
                 StringUtils.trimEmptyToNull(""));
    assertNull("trim empty string to null is null",
StringUtils.trimEmptyToNull(""));
    // but assertNotNull very handy
    assertNotNull("trim x to null is not null",
StringUtils.trimEmptyToNull("x"));
  }
}
```

Example Test Case: PersonTester

The following class contains the JUnit test case for the Person class. The tests are explained in more detail inline. Like the previous example, this class has been created with the JUnit plug-in of JDeveloper.

```
package oracle.code.model.test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import oracle.code.model.Person;

public class PersonTester extends TestCase
{
  public PersonTester(String sTestName)
  {
    super(sTestName);
  }

  /**
   * boolean equals(Person)
   */
  public void testequals()
  {
    // assertions based on objects
    Person jan   = new Person("Jan", new Integer(10));
    Person piet  = new Person("Jan", new Integer(10));
    Person joris = new Person("Jan", new Integer(10));

    Person fred  = new Person();
    Person klaas = new Person("Klaas", new Integer(10));

    // F: assertEquals
    // reflexivity
    assertEquals("jan must equal itself", jan, jan);
    // symmetry
    assertEquals("jan equals piet", jan, piet);
    assertEquals("so piet must equal jan", piet, jan);
    // transitivity
    assertEquals("piet equals joris", piet, joris);
    assertEquals("so jan must equal joris", jan, joris);
    // special cases
    assertEquals("fred equals itself", fred, fred);
    assertFalse("fred not equals null", fred.equals(null));
    assertFalse("jan must not equal klaas", jan.equals(klaas));
```

```
    // G: assertSame for objects
    assertSame("fred is same object as fred", fred, fred);
    assertNotSame("jan is not same object as piet", jan, piet);
  }

}
```

Example Test Suite

The following class is a JUnit test suite that will start the PersonTester and the StringUtilsTester. A test suite can be composed of other test suites, allowing for performing multiple tests at various levels. You could create one super test suite that (indirectly) will execute all unit tests.

The test suite has been created with the JUnit plug-in of JDeveloper, as follows:

- Right-click the project that will contain the tests -> New -> General -> Unit Tests (JUnit) -> Test Suite
- On the 'Class' tab name the class 'AllTests' and name package 'oracle.code.test'. Accept the default Extends ('java.lang.Object').
- On the 'Cases' tab add all test cases by pressing 'Add' and navigate to each individual test case.
- Finish.

When new classes must be added you can either delete the test suite and create a new one, or add the specific test cases manually.

The suite() method adds test cases as a class. Using introspection JUnit is able to extract the test methods from the test case. The name of these methods must therefore start with "test". As an alternate the test methods can be added manually using the addTest() method (instead of addTestSuite()). You can even add the suite() method to your test case and manually add the test methods there. Unless you want to run a specific subset of test methods, the automatic suite extraction is the preferred way. It prevents you from needing to update the suite creation code whenever a test case is added or deleted.

```
package oracle.code.test;

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

public class AllTests
{
  public static Test suite()
  {
    TestSuite suite = new TestSuite("AllTests");

    suite.addTestSuite(oracle.code.model.test.PersonTester.class);

suite.addTestSuite(oracle.code.utils.test.StringUtilsTester.class);

    return suite;
  }

  public static void main(String args[])
  {
```

```
    String args2[] = {"-noloading", "oracle.code.test.AllTests"};

    junit.swingui.TestRunner.main(args2);
  }
}
```

Example Test Fixture

In the following the PersonTester test case is revisited. Two methods have been added: setUp() and tearDown(). JUnit will automatically run the setUp() method before executing each test (read: each distinctive method for which the name starts with 'test') and thus instantiating a test fixture. The tearDown() method is run after each test, destroying the test fixture. When the test case contains one or more test for other methods, the test fixture can than be reused by these tests.

```
package oracle.code.model.test;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import oracle.code.model.Person;

public class PersonTester extends TestCase
{
  private Person jan;
  private Person piet;
  private Person joris;
  private Person fred;
  private Person klaas;

  public PersonTester(String sTestName)
  {
    super(sTestName);
  }

  public void setUp()
  {
    this.jan   = new Person("Jan", new Integer(10));
    this.piet  = new Person("Jan", new Integer(10));
    this.joris = new Person("Jan", new Integer(10));
    this.fred  = new Person();
    this.klaas = new Person("Klaas", new Integer(10));
  }

  public void tearDown()
  {
    this.jan   = null;
    this.piet  = null;
    this.joris = null;
    this.fred  = null;
    this.klaas = null;
  }

  /**
   * boolean equals(Person)
   */
  public void testEquals()
  {
    // F: assertEquals
    // reflexivity
    assertEquals("jan must equal itself", jan, jan);
    // symmetry
    assertEquals("jan equals piet", jan, piet);
    assertEquals("so piet must equal jan", piet, jan);
    // transitivity
    assertEquals("piet equals joris", piet, joris);
    assertEquals("so jan must equal joris", jan, joris);
    // special cases
    assertEquals("fred equals itself", fred, fred);
    assertFalse("fred not equals null", fred.equals(null));
    assertFalse("jan must not equal klaas", jan.equals(klaas));
```

```
    // G: assertSame for objects
    assertSame("fred is same object as fred", fred, fred);
    assertNotSame("jan is not same object as piet", jan, piet);
  }

}
```

Example Unit Testing with ADF Business Components

ADF Test Fixture

The following is an example of how a test fixture can be created that provides a test case with an ADF Application Module. The Application Module in its turn provides a connection to the HR database schema and exposes the View Objects that can be used to store data in or retrieve from the HR schema. Unlike the previous example, this text fixture has been created as a separate class to make it reusable for every test case that needs an ADF Application Module to connect to the database.

This test fixture has been created with the JUnit plug-in of JDeveloper, as follows:

- Right-click the project that will contain the tests -> New -> General -> Unit Tests (JUnit) -> Business Components Text Fixture
- Select the Business Components Project that contains the Application Module you want to use from the dropdown-list
- Select the Application Module from the dropdown-list.
- Select the appropriate configuration.
- Finish.

In the example a configuration has been selected that uses a named JDeveloper connection instead of a JDBC DataSource. This enables running the test case standalone, instead of within the context of a J2EE Web/EJB container.

After creation the class has been modified a little to implement coding practices, like using class variables for static Strings and naming conventions. Furthermore the application module that is returned has the Bundled Exception Mode set to 'true'. As a result failing tests will be bundled and presented all at once when the transaction is committed, instead of failing immediately which would prevent the other steps being executed.

```
package oracle.hr.testfixtures;

import oracle.jbo.*;
import oracle.jbo.client.*;

public class HrServicesConnectFixture
{
  private static final String AM = "oracle.hr.services.HrServices";
  private static final String CF = "HrServicesLocal";

  private ApplicationModule applicationModule;

  public HrServicesConnectFixture()
  {
  }

  /**
   * Acquires an instance of the application module
   */
  public void setUp() throws Exception
  {
```

```
    this.applicationModule =
Configuration.createRootApplicationModule(AM, CF);
    /*
     * Using Bundled Exception Mode, just like the ADF/Struts
support does
     */

this.applicationModule.getTransaction().setBundledExceptionMode(true
);
  }

  /**
   * Cleans up the instance of the application module
   */
  public void tearDown() throws Exception
  {

Configuration.releaseRootApplicationModule(this.applicationModule,
true);
  }

  /**
   * Returns the allocated application module
   *
   * @return Allocated application module
   */
  public ApplicationModule getApplicationModule()
  {
    return this.applicationModule;
  }
}
```

ADF CRUD Test

The following provides an example of how an entity life cycle can be set up for ADF Business Components, using the previous test fixture. In its most elementary form an entity life cycle consists of creating new data in the database, and after that retrieving, updating and finally deleting it.

Especially when the delete fails, the database will be left in a changed state. This is one of the reasons why each test run should start with the same initial setup and why distinctive tests should not depend on the success or failure of other tests, as explained earlier.

Finally, rather than setting the Manager and Location of a Department by using hard-coded values for ManagerId and DepartmentId (as in the example) you could decide to retrieve a Manager and Department from the database based on predetermined names and use the ids of these objects instead.

```
package oracle.hr.dataaccess;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import oracle.hr.testfixtures.HrServicesConnectFixture;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.ViewObject;
import oracle.jbo.Row;

public class DepartmentsTests extends TestCase
{
  private static String DEPARTMENTID = "DepartmentId";
  private static String DEPARTMENTNAME = "DepartmentName";
  private static String MANAGERID = "ManagerId";
  private static String LOCATIONID = "LocationId";
```

```
  HrServicesConnectFixture connectionFixture = new
HrServicesConnectFixture();

  public DepartmentsTests (String sTestName)
  {
    super(sTestName);
  }

  public void testDepartmentsViewObjectLifeCycle()
  {
    ViewObject departmentsView =

this.connectionFixture.getApplicationModule().findViewObject("Depart
ments");
    /**
     * make sure the view object is not null
     */
    assertNotNull(departmentsView);

    /**
     * testing insert of department
     */
    Row departmentRow = departmentsView.createRow();
    try
    {
      departmentRow.setAttribute(this.DEPARTMENTID, "1");
      departmentRow.setAttribute(this.DEPARTMENTNAME, "test
department");
      departmentRow.setAttribute(this.MANAGERID, "100");
      departmentRow.setAttribute(this.LOCATIONID, "1000");
    }
    catch (JboException jbo)
    {
      fail("Should not have thrown this " + jbo.getClass().getName()
+
        " exception yet due to bundled exception mode");
    }

this.connectionFixture.getApplicationModule().getTransaction().commi
t();

    /**
     * testing query of new department
     */
    Row[] departmentRows = new Row[0];
    try
    {
      Key key = departmentRow.getKey();
      departmentRows = departmentsView.findByKey(key, 1);
      assertEquals("expect to find one department with id 1",
departmentRows.length,
                   1);
    }
    catch (JboException jbo)
    {
       fail("Should not have thrown this " +
jbo.getClass().getName() +
        " exception yet due to bundled exception mode");
    }

    /**
     * testing update of the new department
     */
    try
    {
      departmentRows[0].setAttribute(this.DEPARTMENTNAME, "testing
department");

this.connectionFixture.getApplicationModule().getTransaction().commi
t();
    }
    catch (JboException jbo)
```

```
    {
       fail("Should not have thrown this " +
jbo.getClass().getName() +
        " exception yet due to bundled exception mode");
    }

    /**
     * testing deletion of the new department
     */
    try
    {
      departmentRows[0].remove();

this.connectionFixture.getApplicationModule().getTransaction().commi
t();
    }
    catch (JboException jbo)
    {
       fail("Should not have thrown this " +
jbo.getClass().getName() +
        " exception yet due to bundled exception mode");
    }

  }

  public void setUp() throws Exception
  {
    this.connectionFixture.setUp();
  }

  public void tearDown() throws Exception
  {
    this.connectionFixture.tearDown();
  }
}
```

Getting Started With: Unit-testing
December 2004
Author: Jan Kettenis, Remco de Blok
Contributing Authors:

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Copyright © 2003, Oracle. All rights reserved.
This document is provided for information purposes only
and the contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to
any other warranties or conditions, whether expressed orally
or implied in law, including implied warranties and conditions of
merchantability or fitness for a particular purpose. We specifically
disclaim any liability with respect to this document and no
contractual obligations are formed either directly or indirectly
by this document. This document may not be reproduced or
transmitted in any form or by any means, electronic or mechanical,
for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective owners.