Automated testing

lesson3

Good testing is the most important part of software development. It does not matter how beautiful, or fast, or high tech your application is, if it crashes, you leave a poor impression.

Manual testing of the application as an end user, is not a viable way to test, because the real problem is not the code you have written, but the existing code. Usually you test the code you have just written, but you do not retest all the already existing code in your application. And you know that when you touch any part of the code you can break any other part of the application.

While making any changes to your code you want to handle the same with the immense tranquility to save the application from breaking. A way to accomplish this is by using automatic testing. We are going to use automatic testing by means of JUnit.

3.1 JUnit

JUnit[3] is a popular tool for doing automated testing. This tool comes integrated with Eclipse, so you do not need to download it. OpenXava extends the capacities of JUnit allowing you to test an OpenXava module exactly in the same way an end user would. In fact, OpenXava uses HtmlUnit[4], a software that simulates a real browser (including JavaScript) from Java. All this is available from the OpenXava class `ModuleTestBase`. It allows you to automate the test you would do by hand using a real browser in a simple way.

The best way to understand testing in OpenXava is to see it in action.

3.2 ModuleTestBase for testing the modules

The way to create a test for an OpenXava module is by extending the `ModuleTestBase` class from `org.openxava.tests` package. It connects to the OpenXava module as a real browser, and has a lot of methods that allows you to test your module. Let's create the test for your `Customer` module.

3.2.1 The code for the test

Create a new package named `org.openxava.invoicing.tests` and then create a new class named `CustomerTest` inside it with the code as shown in listing 3.1.

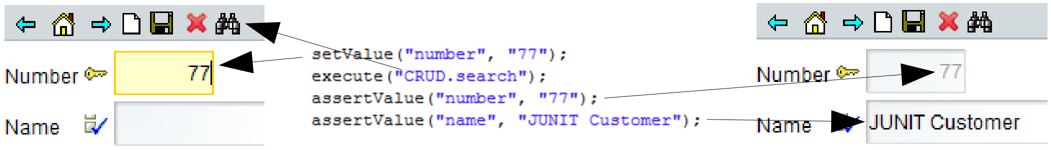
3 http://www.junit.org
4 http://htmlunit.sourceforge.net

Listing 3.1 Automated JUnit test for Customer module

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CustomerTest extends ModuleTestBase { // Must extend from
                                                  // ModuleTestBase
    public CustomerTest(String testName) {
       super(testName,
          "Invoicing",   // We indicate the application name (Invoicing)
          "Customer");   // and the module name (Customer)

    }

    public void testCreateReadUpdateDelete() throws Exception {  // The  test
                                                  // methods must start with 'test'
       // Create
       execute("CRUD.new");   // Clicks on 'New' button
       setValue("number", "77");   // Types 77 as the value for the 'number' field
       setValue("name", "JUNIT Customer");   // Sets the value for the 'name' field
       setValue("address.street", "JUNIT Street");   // Note the dot notation
                                               // to access a reference member
       setValue("address.zipCode", "77555");   // Etc
       setValue("address.city", "The JUNIT city");   // Etc
       setValue("address.state", "The JUNIT state");   // Etc

       execute("CRUD.save");   // Clicks on 'Save' button
       assertNoErrors();   // Verifies that the application does not show errors
       assertValue("number", "");   // Verifies the 'number' field is empty
       assertValue("name", "");   // Verifies the 'name' field is empty
       assertValue("address.street", "");   // Etc
       assertValue("address.zipCode", "");   // Etc
       assertValue("address.city", "");   // Etc
       assertValue("address.state", "");   // Etc

       // Read
       setValue("number", "77");   // Types 77 as the value for the 'number' field
       execute("CRUD.refresh");   // Clicks on 'Refresh' button
       assertValue("number", "77");   // Verifies the 'number' field has 77
       assertValue("name", "JUNIT Customer");   // and 'name' has 'JUNIT Customer'
       assertValue("address.street", "JUNIT Street");   // Etc
       assertValue("address.zipCode", "77555");   // Etc
       assertValue("address.city", "The JUNIT city");   // Etc
       assertValue("address.state", "The JUNIT state");   // Etc

       // Update
       setValue("name", "JUNIT Customer MODIFIED"); // Changes the value
                                                    // of 'name' field
       execute("CRUD.save");   // Clicks on 'Save' button
       assertNoErrors();   // Verifies that the application does not show errors
       assertValue("number", "");   // Verifies the 'number' field is empty
       assertValue("name", "");   // Verifies the 'name' field is empty

       // Verify if modified
       setValue("number", "77");   // Types 77 as the value for 'number' field
       execute("CRUD.refresh");   // Clicks on 'Refresh' button
       assertValue("number", "77");   // Verifies the 'number' field has 77
       assertValue("name", "JUNIT Customer MODIFIED");   // and 'name'
                                         // has  'JUNIT Customer MODIFIED'
```

```
        // Delete
        execute("CRUD.delete");   // Clicks on 'Delete' button
        assertMessage("Customer deleted successfully");   // Verifies that the message
                                              //  'Customer deleted successfully' is shown to the user
    }

}
```

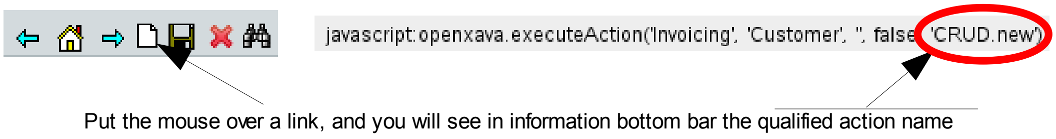
This test creates a new customer, searches it, modifies it, and finally deletes it. You see how you can use methods such as `execute()` or `setValue()` for simulating user actions, and the methods like `assertValue()`, `assertNoErrors()` or `assertMessage()` to verify the state of the user interface. Your test acts as the hands and eyes of the user. Look at figure 3.1.



```
setValue("number", "77");
execute("CRUD.search");
assertValue("number", "77");
assertValue("name", "JUNIT Customer");
```

Figure 3.1 Each test line mimics a user action or verifies the user interface state

In `execute()` you must specify the qualified name of the action, that means `ControllerName.actionName`. How can you know it? Simple, put your mouse over an action link, and you will see in the status bar of your browser a JavaScript code that includes the qualified action name. Just as in figure 3.2.



```
javascript:openxava.executeAction('Invoicing', 'Customer', '', false, 'CRUD.new')
```

Put the mouse over a link, and you will see in information bottom bar the qualified action name

Figure 3.2 Status bar of the browser shows the qualified action name

Now, you know how to create a test for testing the basic CRUD operations of a module. It's not required to write an exhaustive test at first. Just test the basic things, those that you usually test using the browser. Your test will naturally grow with your application and as the user feedback grows.

Let's learn how to execute your test from Eclipse.

3.2.2 *Executing the tests from Eclipse*

As mentioned earlier, JUnit is integrated into Eclipse, so running your tests from Eclipse is plain vanilla. Just put your mouse over the test class, and with the right button choose *Run As > JUnit Test*. Just as in figure 3.3.

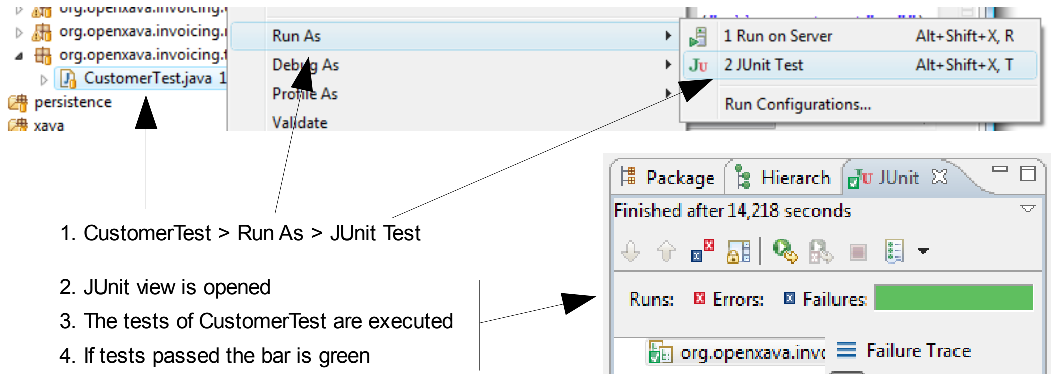


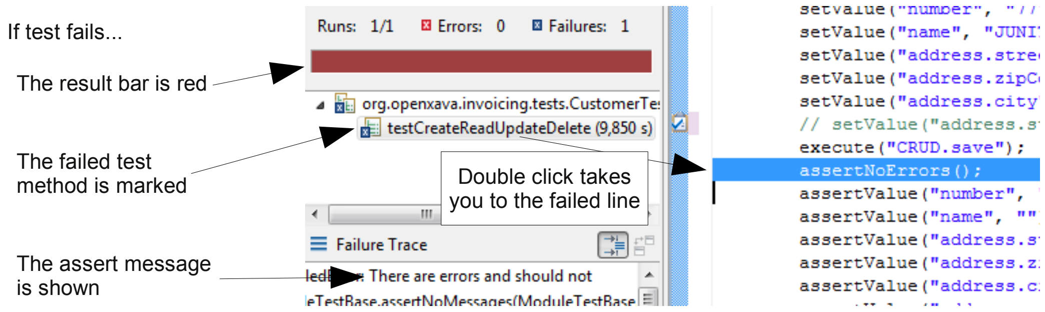
Figure 3.3 Executing a JUnit test from Eclipse

If the test does not pass, the bar will be red. You can try it. Edit the `CustomerTest` and comment the line that sets the value for the `name` field. See listing 3.2.

Listing 3.2 Modifying the test in order to fail

```
...
setValue("number", "77");
// setValue("name", "JUNIT Customer");  // Comment this line
setValue("address.street", "JUNIT Street");
...
```

Now, rerun the test. Since `name` is a required property, an error message will be shown to the user, and the object will not be saved. See figure 3.4.


If test fails...
The result bar is red
The failed test method is marked
The assert message is shown
Double click takes you to the failed line
Runs: 1/1
Errors: 0
Failures: 1
org.openxava.invoicing.tests.CustomerTe
testCreateReadUpdateDelete (9,850 s)
Failure Trace
led: There are errors and should not
eTestBase.assertNoMessages(ModuleTestBase

```
setValue("number", "77
setValue("name", "JUNI
setValue("address.stre
setValue("address.zipC
setValue("address.city
// setValue("address.s
execute("CRUD.save");
assertNoErrors();
assertValue("number",
assertValue("name", ""
assertValue("address.s
assertValue("address.z
assertValue("address.c
```

Figure 3.4 Failed test executed from Eclipse

The failed assert is `assertNoErrors()`, which in addition to failure shows the error on the console. So, in the execution console of your test you will see the message in listing 3.3.

Listing 3.3 Console message when assertNoErrors() fails

```
16-jul-2009 18:03 org.openxava.tests.ModuleTestBase assertNoMessages
SEVERE: Error unexpected: Value for Name in Customer is required
```

The problem is clear. The customer is not saved because the name is required, and it's not specified.

You have seen how the test behaves when it fails. Now, you can uncomment back the guilty line and run the test again to verify that everything is OK.

3.3 Adding the JDBC driver to the Java Build Path

Surely you have noted a clumsy stack trace in the console view on the execution of your test, somewhat like listing 3.4.

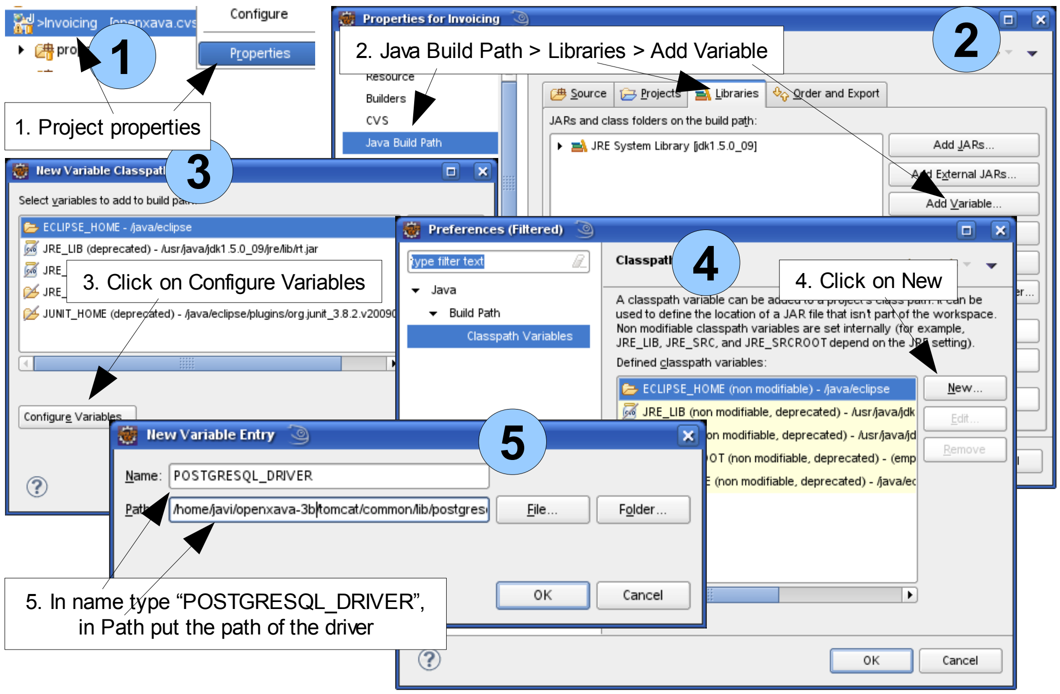
Listing 3.4 Exception “JDBC Driver not found” executing the junit test

```
 javax.persistence.PersistenceException:
 [PersistenceUnit: junit] Unable to build EntityManagerFactory
     at ...
 Caused by:
 org.hibernate.HibernateException: JDBC Driver class not found:
org.postgresql.Driver
     at ...
```

In spite of this exception the test works fine. OpenXava test tries to connect to the dababase in order to obtain some metadata, if it does not get connected then it tries to obtain the metadata in other ways, so finally the test works.

We are going to add the PostgreSQL JDBC driver to the Java Build Path of your project. In this way we'll avoid the clumsy stack trace, and more importantly, your test can access the database directly. This is very important, because it is a common case whether you want to use JPA or JDBC directly inside your test.

To add the PostgreSQL JDBC driver to the Java Build Path of your project follow the instruction in figure 3.5.


1. Project properties
Configure
Properties
2. Java Build Path > Libraries > Add Variable
Properties for Invoicing
Resource
Builders
CVS
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
JRE System Library [jdk1.5.0_09]
Add JARs...
Add External JARs...
Add Variable...
3. Click on Configure Variables
New Variable Classpath
Select variables to add to build path:
ECLIPSE_HOME - /java/eclipse
JRE_LIB (deprecated) - /usr/java/jdk1.5.0_09/jre/lib/rt.jar
JUNIT_HOME (deprecated) - /java/eclipse/plugins/org.junit_3.8.2.v20090
Configure Variables...
4. Click on New
Preferences (Filtered)
type filter text
Java
Build Path
Classpath Variables
A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. Non modifiable classpath variables are set internally (for example, JRE_LIB, JRE_SRC, and JRE_SRCROOT depend on the JRE setting).
Defined classpath variables:
ECLIPSE_HOME (non modifiable) - /java/eclipse
JRE_LIB (non modifiable, deprecated) - /usr/java/jdk
New...
Edit...
Remove
OK
Cancel
5. In name type “POSTGRESQL_DRIVER”, in Path put the path of the driver
New Variable Entry
File...
Folder...
OK
Cancel

Name: POSTGRESQL_DRIVER

Path: /home/javi/openxava-3b/tomcat/common/lib/postgres

Figure 3.5 Adding PostgreSQL JDBC driver to the project Java Build Path

To finish this, just close all the dialogs by clicking on OK. In this case we have used a classpath variable to add the library to the project.

From now on your JUnit tests can connect to your PostgreSQL database.

3.4 Creating test data using JPA

In your first test, `CustomerTest`, the test itself starts creating the data that is used in the rest of the test. This is a good way to go, especially if you want to test the data entry functionality too. But, if you want to test only a small buggy case, or your module simply does not allow adding new objects, you can create the data you need for testing using JPA from your test.

3.4.1 Using setUp() and tearDown() methods

We are going to use `ProductTest` to learn how to use JPA for creating test data. We'll create some products before each test execution and remove them afterwards. Let's see the code for `ProductTest` in listing 3.5.

Listing 3.5 Creating and removing the testing data for each test

```
package org.openxava.invoicing.tests;

import java.math.*;

import org.openxava.invoicing.model.*;
import org.openxava.tests.*;
import static org.openxava.jpa.XPersistence.*;

public class ProductTest extends ModuleTestBase {

    private Author author;   // We declare the entities to be created
    private Category category;   // as instance members in order
    private Product product1;   // to be available from inside any test method
    private Product product2;   // and to be removed at the end of each test

    public ProductTest(String testName) {
        super(testName, "Invoicing", "Product");
    }

    protected void setUp() throws Exception {  // setUp() is always executed
                                                 // before each test
        createProducts();   // Creates the data used in the tests
        super.setUp();   // It's needed because ModuleTestBase uses it for initializing
    }

    protected void tearDown() throws Exception {    // tearDown() is always executed
                                                     // after each test
        super.tearDown();   // It's needed, ModuleTestBase  closes resources here
        removeProducts();   // The data used for testing is removed
    }

    public void testRemoveFromList() throws Exception { … }

    public void testChangePrice() throws Exception { … }

    private void createProducts() { … }

    private void removeProducts() { … }

}
```

Here we are overwriting the `setUp()` and `tearDown()` methods. These methods are JUnit methods that are executed just before and after each test execution. We create the testing data before each test execution, and remove the data after each test execution. Thus, each test can rely on the precise data to be executed. It does not matter if some other test removes or modifies the data, or the execution order of the test. Always, at the beginning of each test we have all the data ready to use.

3.4.2 Creating data with JPA

The `createProducts()` method is responsible for creating the test data using JPA. Let's examine it in listing 3.6.

Listing 3.6 Creating test data using JPA

```
private void createProducts() {
    // Creating the Java objects
    author = new Author();   // Regular Java objects are created
    author.setName("JUNIT Author");   // We use setters just as in plain Java
    category = new Category();
    category.setDescription("JUNIT Category");

    product1 = new Product();
    product1.setNumber(900000001);
    product1.setDescription("JUNIT Product 1");
    product1.setAuthor(author);
    product1.setCategory(category);
    product1.setPrice(new BigDecimal("10"));

    product2 = new Product();
    product2.setNumber(900000002);
    product2.setDescription("JUNIT Product 2");
    product2.setAuthor(author);
    product2.setCategory(category);
    product2.setPrice(new BigDecimal("20"));

    // Marking as persistent objects
    getManager().persist(author);   // getManager() is from XPersistence
    getManager().persist(category);   // persist() marks the object as persistent
    getManager().persist(product1);   // so it will be saved to the database
    getManager().persist(product2);

    // Commit changes to the database
    commit();    // commit() is from XPersistence. It saves all object to the database
                 // and commits the transaction
}
```

As you can see, first you create the Java objects in the Java conventional way. Note that you assign it to instance members. Thus you can use it inside tests. Then, you mark them as persistent, using the `persist()` method of the JPA `EntityManager`. To obtain the `PersistenceManager` you only have to write `getManager()` because you have the static import above (listing 3.7).

Listing 3.7 Static import for easier use of getManager() and commit()

```
import static org.openxava.jpa.XPersistence.*;
...
getManager().persist(author);
    // Thanks to the XPersistence static import it's the same as
XPersistence.getManager().persist(author);
...
commit();
    // Thanks to the XPersistence static import it's the same as
XPersistence.commit();
```

To finalize, `commit()` (also from `XPersistence`) saves all the data from object to database and then commits the transaction. After that, the data is in the database ready to be used by your test.

3.4.3 *Removing data with JPA*

After the test is executed we remove the test data in order to leave the database clean. This is done by the `removeProducts()` method. See it in listing 3.8.

Listing 3.8 removeProducts()/remove() are responsible for removing test data

```
private void removeProducts() {   // Called from tearDown() so it's executed after each test
    remove(product1, product2, author, category);   // remove() removes
    commit();   // Commits the changes to database, in this case deleting  data
}

private void remove(Object ... entities) {   // Using varargs argument
    for (Object entity: entities) {   // Iterating for all arguments
       getManager().remove(getManager().merge(entity));   // Removing(1)
    }
}
```

It's a simple loop to remove all the entities used in the test. To remove an entity in JPA you have to use the `remove()` method, but in this case you have to use the `merge()` method too (shown as 1). This is because you cannot remove a detached entity. When you use `commit()` in `createProducts()` all saved entities become detached entities. This is because they continue being valid Java object but the persistent context (the union between entities and database) has been lost on `commit()`, so you must reattach them to the new persistent context. This concept is easy to understand seeing the listing 3.9 code.

Listing 3.9 Using merge() to reattach detached instances

```
getManager().persist(author);   // author is attached to the current persistence context
commit();   // The current persistence context is over, so author becomes detached

getManager().remove(author);   // It fails because author is detached

author = getManager().merge(author);   // Reattaches author to the current  context
getManager().remove(author);   // It works
```

Apart from this curious detail about `merge()`, the code for removing is just simple.

3.4.4 *Filtering data from list mode in a test*

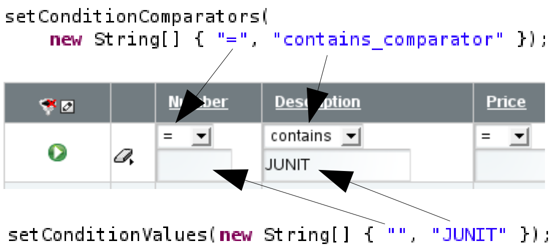
Now that you know how to create and remove the data for the tests, let's examine the test methods for your `Product` module. The first one is `testRemoveFromList()` that checks a row in list mode and clicks on “Delete selected” button. Let's see the code in listing 3.10.

Listing 3.10 Testing list mode doing a data filter

```
public void testRemoveFromList() throws Exception {
    setConditionValues(   // Put the values for filtering data
       new String[] { "", "JUNIT" });
```

```
        setConditionComparators(    // Put the comparators for filtering data
            new String[] { "=", "contains_comparator" });
        execute("List.filter");    // Clicks on filter button
        assertListRowCount(2);    // Verifies that there are 2 rows
        checkRow(1);    // We select row 1 (really the second one)
        execute("CRUD.deleteSelected");    // Clicks on the delete button
        assertListRowCount(1);    // Verifies that now there is only 1 row
    }
```

Here we filter in list mode all products that contain the “JUNIT” word (remember that you have created two of them in `createProducts()` method), then we verify that there are two rows, select the second product, and remove it, verifying at the end that one product remains in the list.


```
setConditionComparators(
    new String[] { "=", "contains_comparator" });
```

```
setConditionValues(new String[] { "", "JUNIT" });
```

Figure 3.6 Filtering list in a test

You have learned how to select a row (using `checkRow()`) and how to assert the row count (using `assertListRowCount()`). The trickiest part might be filtering the list using `setConditionValues()` and `setConditionComparators()`. Both methods receive an array of strings with values and comparators for the condition, just as shown in figure 3.6. The values are assigned to the list filter user interface sequentially (from left to right). You do not need to specify all values. The `setConditionValues()` method accepts any string value whereas `setConditionComparators()` accepts the next possible values: starts_comparator, contains_comparator, =, <>, >=, <=, > and <.

3.4.5 Using entity instances inside a test

The remaining test, `testChangePrice()`, simply chooses a product and changes its price. We are going to use it in an entity created in `createProducts()`. The code is in listing 3.11.

Listing 3.11 Testing using a value from an entity object created for the test

```
public void testChangePrice() throws Exception {
    // Searching the product1
    execute("CRUD.new");
    setValue("number", Integer.toString(product1.getNumber()));    // (1)
    execute("CRUD.refresh");
    assertValue("price", "10.00");

    // Changing the price
    setValue("price", "12.00");
    execute("CRUD.save");
    assertNoErrors();
    assertValue("price", "");
    // Verifying
    setValue("number", Integer.toString(product1.getNumber()));  // (1)
    execute("CRUD.refresh");
    assertValue("price", "12.00");
}
```

The only new thing in this test is to provide a value to number to search the product. We get the value using `product1.getNumber()` (shown as 1). Remember that `product1` is an instance variable of the test that is populated in `createProducts()`, which is called from `setUp()` so it is executed before each test.

You have a test class for `Product` and at the same time you have learned testing with test data created using JPA.

3.5 Using existing data for testing

Sometimes you can simplify the test by relying on a test database which is populated with the data needed for testing. If you do not want to test data creation from the module itself, and you do not remove data in the test, this can be a good option.

For example, you can test `Author` and `Category` with a simple test like the one shown in listing 3.12.

Listing 3.12 Test relying on existing data in the database

```
public class AuthorTest extends ModuleTestBase {

    public AuthorTest(String testName) {
       super(testName, "Invoicing", "Author");
    }

    public void testReadAuthor() throws Exception {
       assertValueInList(0, 0, "JAVIER CORCOBADO");  // The first author
                                                  // in the list is JAVIER CORCOBADO
       execute("Mode.detailAndFirst"); // On change detail mode the
                                       // the first object in list is displayed
       assertValue("name", "JAVIER CORCOBADO");
       assertCollectionRowCount("products", 2);  // It has 2 products
       assertValueInCollection("products", 0,   // Row  0 of products
             "number", "2");  // has “2” in “number” column
       assertValueInCollection("products", 0,
             "description", "Arco iris de lágrimas");
       assertValueInCollection("products", 1, "number", "3");
       assertValueInCollection("products", 1,
             "description", "Ritmo de sangre");
    }

}
```

This test verifies that the first author in the list is “JAVIER CORCOBADO”. It goes to the detail and asserts that it has a collection called products with 2 products: “Arco iris de lágrimas” and “Ritmo de sangre”. By the way, now you have learned how to use `assertValueInList()`, `assertValueInCollection()` and `assertCollectionRowCount()` methods.

We can use the same technique to test the `Category` module. Let's see it in listing 3.13.

Listing 3.13 Testing only list mode with existing data

```
public class CategoryTest extends ModuleTestBase {

    public CategoryTest(String testName) {
       super(testName, "Invoicing", "Category");
    }

    public void testCategoriesInList() throws Exception {
       assertValueInList(0, 0, "MUSIC");  // Row 0 column 0 has “MUSIC”
       assertValueInList(1, 0, "BOOKS");  // Row 1 column 0 has “BOOKS”
       assertValueInList(2, 0, "SOFTWARE");  // Row 2 column 0 has “SOFTWARE”
    }

}
```

In this case we see that in the list the first three categories are “MUSIC”, “BOOKS” and “SOFTWARE”.

You have seen how the technique of using preexisting data from a test database allows you to create simpler tests. Starting from a simple test and further complicating it on demand is a good way to go.

3.6 Testing collections

Now it's time to face the test for the main module of your application, the `InvoiceTest`. As of now the functionality of the `Invoice` module is limited. You can only add, remove and modify invoices. Even so, this is a big test. It contains a collection, so you will learn here how to test collections.

3.6.1 Breaking down tests in several methods

Listing 3.14 shows the `InvoiceTest` code.

Listing 3.14 Test for creating an Invoice is broken down into several methods

```
package org.openxava.invoicing.tests;

import java.text.*;
import java.util.*;
import javax.persistence.*;
import org.openxava.tests.*;
import org.openxava.util.*;

import static org.openxava.jpa.XPersistence.*;  // To use JPA

public class InvoiceTest extends ModuleTestBase {

    private String number;  // To store the number of the tested invoice

    public InvoiceTest(String testName) {
       super(testName, "Invoicing", "Invoice");
    }

    public void testCreateInvoice() throws Exception {  // The test method
       verifyDefaultValues();
       chooseCustomer();
       addDetails();
       setOtherProperties();
       save();
       verifyCreated();
       remove();
    }

    private void verifyDefaultValues() throws Exception { … }

    private void chooseCustomer() throws Exception { … }

    private void addDetails() throws Exception { … }

    private void setOtherProperties() throws Exception { … }

    private void save() throws Exception { … }

    private void verifyCreated() throws Exception { … }

    private void remove() throws Exception { … }

    private String getCurrentYear() { … }

    private String getCurrentDate() { … }

    private String getNumber() { … }

}
```

The only test method in this class is `testCreate()`, but because this test is somewhat large, it is better to break it down into several shorter methods. In fact, it's a good OO[5] practice to write short methods.

Because this method is short you can see in a glance what it does. In this case the method verifies the default values for a new invoice, chooses a customer, adds the details, adds other properties, saves the invoice, verifies that it is correctly saved and finally deletes it. Let's dip into the details of these steps.

5 Object Oriented

3.6.2 *Asserting default values*

First, it verifies whether the default values for a new invoice are correctly calculated or not. This is done by the `verifyDefaultValues()` method. It is in listing 3.15.

Listing 3.15 Verifying default values on creation of a new invoice

```
private void verifyDefaultValues() throws Exception {
    execute("CRUD.new");
    assertValue("year", getCurrentYear());
    assertValue("number", getNumber());
    assertValue("date", getCurrentDate());
}
```

When the user clicks on “New”, the year, number and date field must be prefilled with valid data. The `verifyDefaultValues()` method tests this. It uses several utility methods to calculate the expected values. You can see them in listing 3.16.

Listing 3.16 Utility methods used by verifyDefaultValues()

```
private String getCurrentYear() {  // Current year in string format
    return new SimpleDateFormat("yyyy").format(new Date());  // The  standard way
                                                            // to do it with Java
}

private String getCurrentDate() {  // Current date as string in short format
    return DateFormat.getDateInstance(  // The standard way to do it with Java
       DateFormat.SHORT).format(new Date());
}

private String getNumber() {  // The invoice number for a new invoice
    if (number == null) {  // We use lazy initialization
       Query query = getManager().  // A JPA query to get the last number
          createQuery(
    "select max(i.number) from Invoice i where i.year = :year"
          );
       query.setParameter("year",
          Dates.getYear(new Date()));  // Dates is an OpenXava utility
       Integer lastNumber = (Integer) query.getSingleResult();
       if (lastNumber == null) lastNumber = 0;
       number = Integer.toString(lastNumber + 1);  // Adding 1 to the last
                                                   // invoice number
    }
    return number;
}
```

The `getCurrentYear()` and `getCurrentDate()` methods use classic Java techniques to format the date as string.

The `getNumber()` method, on the other hand, is a little more complex. It uses JPA to calculate the last invoice number of the current year, then return this value plus one. Due to its access to the database it is heavier than a simple Java calculation, therefore we use lazy initialization. Lazy initialization delays the calculation until the first time it is needed, and stores it for future use. We do it by saving the value in the `number` field.

Note the usage of the `Dates` class to extract the year from the date. `Dates` is a utility class you can find in `org.openxava.util`.

3.6.3 Data entry

Now it's time for the `chooseCustomer()` method of the invoice. See the code in listing 3.17.

Listing 3.17 Testing choosing a customer for the invoice

```
private void chooseCustomer() throws Exception {
    setValue("customer.number", "1");
    assertValue("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");
}
```

Upon entry of the customer number the customer name is simply filled with the appropriate value. With this the customer 1 is associated to the current invoice.

And now comes the most tricky part of the test: adding the detail lines. You have the code for `addDetails()` in listing 3.18.

Listing 3.18 Testing adding elements to a collection

```
private void addDetails() throws Exception {
    // Adding a detail line
    assertCollectionRowCount("details", 0);  // The collection is empty
    execute("Collection.new",   // Clicks on new button to add a new element
       "viewObject=xava_view_details");    // viewObject is needed to
                                           // determine what collection is affected
    setValue("product.number", "1");
    assertValue("product.description",
       "Peopleware: Productive Projects and Teams");
    setValue("quantity", "2");
    execute("Collection.saveAndStay");  // Saves the collection element and
                                        // it does not close the dialog
    assertNoErrors();  // No errors on save of the detail

    // Adding another detail
    setValue("product.number", "2");
    assertValue("product.description", "Arco iris de lágrimas");
    setValue("quantity", "1");
    execute("Collection.save");  // Saves the collection element and closes the dialog
    assertNoErrors();
    assertCollectionRowCount("details", 2);  // Now we have 2 rows
}
```

Testing a collection is the same as testing any other part of your application. You have to follow the same steps as an end user with a browser. Note that you have to use `viewObject=xava_view_details` as an argument for some collection actions.

Now that we have the details added, we are going to fill the remaining data and save the invoice. The remaining data is set by `setOtherProperties()` method (listing 3.19).

Listing 3.19 Assigning values to the remaining properties before saving

```
private void setOtherProperties() throws Exception {
    setValue("remarks", "This is a JUNIT test");
}
```

Here we give a value to the `remarks` field. Now we are ready to save the invoice. See it in listing 3.20.

Listing 3.20 Saving the invoice from the JUnit test

```
private void save() throws Exception {
    execute("CRUD.save");
    assertNoErrors();

    assertValue("customer.number", "");
    assertCollectionRowCount("details", 0);
    assertValue("remarks", "");
}
```

It simply clicks on “Save”, then verifies for any errors and makes sure that the view is clean.

3.6.4 Verifying the data

Now, we will search the newly created invoice to verify that it has been saved correctly. This is done by the `verifyCreated()` method that you can see in listing 3.21.

Listing 3.21 Verifying that the invoice data has been saved correctly

```
private void verifyCreated() throws Exception {
    setValue("year", getCurrentYear());   // The current year to year field
    setValue("number", getNumber());   // The invoice number of the test
    execute("CRUD.refresh");   // Load the invoice back from the database

    // In the rest of the test we assert that the values are the correct ones
    assertValue("year", getCurrentYear());
    assertValue("number", getNumber());
    assertValue("date", getCurrentDate());

    assertValue("customer.number", "1");
    assertValue("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");

    assertCollectionRowCount("details", 2);

    // Row 0
    assertValueInCollection("details", 0, "product.number", "1");
    assertValueInCollection("details", 0, "product.description",
       "Peopleware: Productive Projects and Teams");
    assertValueInCollection("details", 0, "quantity", "2");
    // Row 1
    assertValueInCollection("details", 1, "product.number", "2");
    assertValueInCollection("details", 1, "product.description",
       "Arco iris de lágrimas");
    assertValueInCollection("details", 1, "quantity", "1");

    assertValue("remarks", "This is a JUNIT test");
}
```

After searching the created invoice we verify whether the values we have saved are there. If the test reaches this point your `Invoice` module works fine. The only thing remaining is to delete the created invoice so that the test can be executed again. We do that in the `remove()` method (listing 3.22).

Listing 3.22 Removing the Invoice used for the test

```
private void remove() throws Exception {
    execute("CRUD.delete");
    assertNoErrors();
}
```

It just clicks on “Delete” and verifies that no errors are produced.

Congratulations! You have your `InvoiceTest` completed.

3.7 Suite

You have 5 test cases to preserve the quality of your application. When you finish some enhancement or fix of your application you must execute all your tests to ensure that your already existing functionality is not broken.

Traditionally, to execute all the test for your application you have to create a test suite, and execute it. A test suite is a class that aggregates all your JUnit tests so you can execute them all at once. Fortunately, if you are working with Eclipse you do not need to write a test suite class, Eclipse allows you to execute all the test for your application automatically, as shown in figure 3.7.

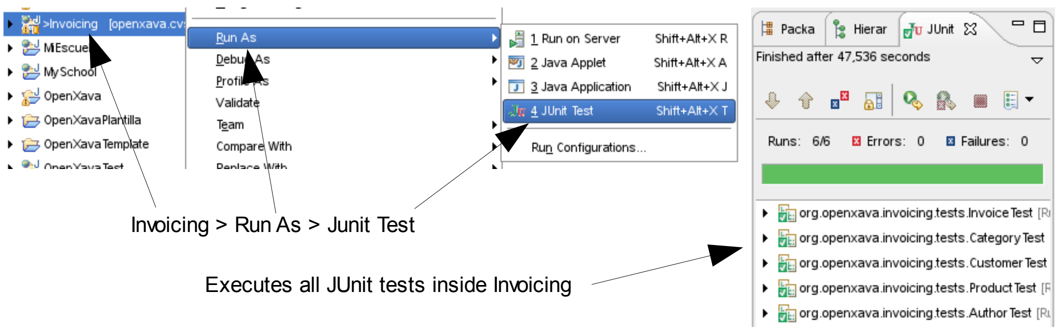

>Invoicing [openxava.cv
MEscuela
MySchool
OpenXava
OpenXavaPlantilla
OpenXavaTemplate
Run As
Debug As
Profile As
Validate
Team
Compare With
1 Run on Server Shift+Alt+X R
2 Java Applet Shift+Alt+X A
3 Java Application Shift+Alt+X J
4 JUnit Test Shift+Alt+X T
Run Configurations...
Packa
Hierar
JUnit
Finished after 47,536 seconds
Runs: 6/6
Errors: 0
Failures: 0
org.openxava.invoicing.tests.InvoiceTest
org.openxava.invoicing.tests.CategoryTest
org.openxava.invoicing.tests.CustomerTest
org.openxava.invoicing.tests.ProductTest
org.openxava.invoicing.tests.AuthorTest
Invoicing > Run As > Junit Test
Executes all JUnit tests inside Invoicing

Figure 3.7 Executing all the tests for the project at once

That is, if you execute *Run As > JUnit Test* on the project then all its JUnit tests are executed.

3.8 Summary

You have automated the tests for all the current functionality of your application. This test code seems to be more verbose and boring than the real application code. But remember, the test code is the most valuable asset you have. Right now you may not believe me, but try to do tests and once they save your life, you will not develop without test code any more.

What to test? Don't do an exhaustive test at first. It's better to test a little than to test nothing. If you try to do exhaustive testing you will end up testing nothing. Start doing a little testing of all your code, and with any new feature or fix also write the test for it. In the end, you will have a very powerful test suite. Test little but test always.

In fact, testing is an on going task. In order to preach with the example, from now on we'll write all the tests for the code we will develop in the rest of the book. Thus you will learn more tips about testing in the next lessons.