

## **PM - Workshop 2**

### **Testing, Debugging and GUI building**

Emil Djupfeldt  
Project Course IT (TDA366/DIT211)  
Chalmers Tekniska Högskola  
Mars 2010

## ***Introduction***

The purpose of this workshop is to give an introduction to some of the tools available in the Eclipse IDE. Unit testing, dependency checking, debugging and using the gui builder will be covered.

## ***Initial setup***

Before you get started you will need to create an empty project in Eclipse. However, since we will be writing tests later as well as ordinary code the project should contain an additional source folder called *test*.

You can do this by selecting Java Project from the File -> New menu. On the first page that shows up fill in an appropriate project name (i.e. tda366-debug) and press next.

On the second page you will be given an option to create additional folders. It's a good idea to keep test code separate from the rest of the source. On the Source tab click Create new source folder and give it the name *test*.

You should now have a folder named *src* and a folder named *test*. Press Finish to create the project.

Now create a class called *MathModel* in your project. On the page that shows up when creating a new class make sure it will be placed in the *src* folder in your project, and not the *test* folder.

Once you have created the class enter the following code:

```
public class MathModel {
    public int multiply(int a, int b) {
        return a / b; // Incorrect on purpose
    }
}
```

This is a simple program with an glaringly obvious bug. The bug will be dealt with later so keep it around for now.

## JUnit tests

To assure that a method or class behaves in the intended manner it is often beneficial to write some code to test it. JUnit tests provide a standardised framework for this purpose.

Right click on your *MathModel* class in the package explorer and create a new JUnit Test Case. Make sure JUnit 4 test is selected instead of JUnit 3, and change the source folder to the *test* folder you created before. On the second page you will get a choice of which methods you want to test. Select *multiply(int, int)* from *MathModel* and press Finish.

If this is the first unit test you create in your project you might get a dialog asking if you want to add JUnit 4 to your build path. Make sure the option to do so is selected and press OK.

Eclipse has now created a class *MathModelTest* for you and a method *testMultiply()* in it. Currently the test will always fail. Change it to look like this:

```
@Test
public void testMultiply() {
    MathModel m = new MathModel();

    assertTrue(m.multiply(2, 5) == 10);
}
```

The *@Test* attribute identifies this method as a test case. There are a few more attributes that can be used in a unit test, some of which will be covered later.

The *assertTrue()* method is what makes the test case work. It takes one boolean as argument. If the value is false the the test case fails and if it is true it succeeds. There are many more assert methods but in this case *assertTrue()* is sufficient.

If you feel that just one check is not enough to guarantee that the *multiply()* method works as intended you can add another *assertTrue()* with a different condition after the current one.

Now your test is ready to be used. Run it by right clicking on the class in the package explorer and selecting Run As -> JUnit Test. When the test has finished running you should get a report of the test cases. As you have only one test and the *multiply(int, int)* method is flawed all tests should fail.

Fix the bug in *MathModel* and then run the unit test again. Now all tests should report ok.

Sometimes several tests operate on the same or similar objects. In this case a fixture can be created to simplify the test cases. Change *MathModelTest* to look like the code on the next page.

The method identified by *@Before* will be run before any test case is executed and the method identified by *@After* will be run after the test cases. Since having a fixture for only one test case is a bit superfluous you should expand the *MathModel* class with more methods and create test cases for each of them.

```
import static org.junit.Assert.*;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class MathModelTest {
    private MathModel m;

    @Before
    public void init() {
        m = new MathModel();
    }

    @After
    public void uninit() {
        m = null;
    }

    @Test
    public void testMultiply() {
        assertTrue(m.multiply(2, 5) == 10);
    }
}
```

## Jigloo GUI builder

Now that you have a working (although simple) math class it is time to create a GUI for it. Select Other from the File -> New menu. In the window that appears browse to GUI Forms -> Swing -> JFrame and press next. Enter MathWindow as name, make sure that Add main method is checked and press Finish.

Since this is a simple math program you will need a way to input values and a way to display the results. An easy way to do this is to add two text fields and a label. However, before this can be done you will need to set the layout of the frame so the components in it will be displayed correctly.

From the top of the gui editor select the Layout tab. Then press the *GroupLayout* button in the toolbar that appeared and finish by clicking somewhere inside the frame below. This will set the current layout for the frame to a *GroupLayout*.

If you get a question to add the layout library to your project the press Yes since you will need this to run your project.

Next step is to add the text fields. Select the Components tab and press the *JTextField* button. If you move the mouse around inside the frame you will notice that one or more red dotted lines show up sometimes. These are guides to help place the components at equal distances. Place the text field inside the frame by clicking where you want it to be. In the dialog that shows up you can enter a name and default value for the text field. Since the first argument to the *multiply()* method was called *a* you can call the text field *txtA*. An appropriate default value could be 0.

Since *multiply()* takes two arguments, create another text field called *txtB*.

For the result you should create a label. Locate the *JLabel* button in the toolbar and place one somewhere in the frame. You can call it *lblResult*.

Now you have a way to input values and display the results. However, the program also needs to know when to update the result. For this you need a button. Locate the *JButton* button in the toolbar and place one in the frame.

To connect the button to the code you will need an *AbstractAction*. It can be found in the More Components tab. Press the *AbstractAction* and then click outside the frame to add it as a non-visual component. Find the *AbstractAction* in the Outline pane and drag it to the button you added before. This will connect the action to the button.

Now switch over to the source view. You can see the autogenerated code for the frame. The interesting parts are the the constructor and the *actionPerformed()* method.

Create a private instance variable called *model* to contain the *MathModel* and update the constructor to take it in as an argument. Then update the main method to create the *MathModel* and pass it along to the constructor.

Then locate the *actionPerformed()* method inside the *AbstractAction* close to the bottom and change it to look like this:

```
public void actionPerformed(ActionEvent evt) {
    int a = Integer.parseInt(txtA.getText());
    int b = Integer.parseInt(txtB.getText());

    int result = model.multiply(a, b);

    lblResult.setText("" + result);
}
```

Now you should be able to run your frame. Enter a few values and try calculating the results. Since the program only multiplies the values it's not very useful. Go back to the gui editor and add a method to select the other operations you added to *MathModel*. Try using a list, a combobox or

radio buttons. Then go back and make the corresponding changes to the code.

## Debugging

A very useful method to find out why something is not working as intended in a program is to debug it. One possible method is to singlestep through every line in the program but this can get tedious for everything but the smallest code bases. To avoid this it is possible to add breakpoints to the program. A breakpoint is a condition that when fulfilled pauses the execution of the program and transfers control to the debugger. A common condition is to break on a certain line in a source file.

To add a break point to your program, right click on the thick gray edge to the left of the source on the first line in the *actionPerformed()* method in the *MathWindow* class and choose Toggle Breakpoint from the menu. Then start the debugger either by pressing the buglike button next to the run button in the toolbar or by choosing Debug from the Run menu.

The program will start as usual but once you press the button to calculate the result the breakpoint will trigger and Eclipse will enter the debug perspective. Here you will get an overview of the variables available at the current location in the code and their values, a list of the threads in the program and the call stack for each of them, and of course the current piece of code.

You can step through the code using F5 to step into functions and expressions, F6 to step to the next line and F8 to resume the program. You can also change values of variables. Try stepping with F6 until you get to the line where it calls a method from the model and then step with F5 to enter it. Once in the model code you can continue stepping with either F5 or F6 and it will go back to *actionPerformed()* once the method returns.

Once back in *actionPerformed()* step to the line where the label text is set. Before executing this line, find the variable containing the result in the Variables pane and change its value by clicking on it and entering something else. Then step past the text update line and then press F8 to resume the program. Notice how the result does not match up with the input.

Another thing to take notice of is the *this* variable in the Variables pane. Since *actionPerformed()* is located inside an inner class *this* points to the inner class and not *MathWindow*. However, if you expand the *this* variable you will find a member variable called *this\$0*. This one refers to the outer class and contains all its instance variables.

## **JDepend**

JDepend is a tool to check dependencies between packages in java. Combined with unit tests it can also be used to enforce those dependencies.

Before you begin you will need to download and add the JDepend library to your project. It can be found at <http://www.clarkware.com/software/jdepend-2.9.zip>

To add the library, go to the Project -> Properties menu, navigate to Java Build Path and choose the Libraries tab. From there, click Add external JARs... and locate jdepend-2.9.jar.

Before a unit test that checks the dependencies of the project can be constructed JDepend needs some setup. More specific, it needs to know where the compiled project code is located and what packages to ignore when checking dependencies. Normally you only want to ignore the default java api packages.

After setting everything up you have to decide which other packages the packages in your code are allowed to depend on.

The only package in this workshop is the default package, and the packages used so far are *org.junit* and *org.jdesktop.layout*.

None of the imported packages depend on anything but the java api, but the default package in the project depends on all the others.

With all this in mind, change MathModelTest to look like the code on the next page. Remember to change the path to the project. Then run the unit test again.

If everything went well both the unit tests should report success. However, the dependency will fail. This is because when introducing the JDepend framework a dependency for the *jdepend.framework* package was added.

Change the code to include this dependency as well.

```

import static org.junit.Assert.*;

import java.io.IOException;

import jdepend.framework.*;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class MathModelTest {
    private MathModel m;
    private JDepend dep;

    @Before
    public void init() throws IOException {
        m = new MathModel();

        PackageFilter filter = new PackageFilter();
        filter.addPackage("java.*");
        filter.addPackage("javax.*");

        dep = new JDepend(filter);
        dep.addDirectory("/Users/egladil/src/tda366-debug/bin");
    }

    @After
    public void uninit() {
        m = null;
        dep = null;
    }

    @Test
    public void testMultiply() {
        assertTrue(m.multiply(2, 5) == 10);
    }

    @Test
    public void testDependencies() {
        DependencyConstraint constraint = new DependencyConstraint();

        JavaPackage junit = constraint.addPackage("org.junit");
        JavaPackage jdesktop = constraint.addPackage("org.jdesktop.layout");
        JavaPackage def = constraint.addPackage("Default");

        def.dependsUpon(junit);
        def.dependsUpon(jdesktop);

        dep.analyze();

        assertTrue("Dependency mismatch", dep.dependencyMatch(constraint));
    }
}

```

## ***Further reading***

If you want additional information on testing and a great example of how something seemingly trivial can require a quite extensive test suite you should read this post about the “triangle example”:

<http://blog.chilly.ca/?p=194>

Information about Jigloo can be found at their web page. There you can also find information on how to install it in case your copy of Eclipse did not include it by default.

<http://www.cloudgarden.com/jigloo/>

JDepend can be used in several more ways in addition to the one explained here. There is for example a way to list dependencies and also a GUI to view them.

<http://clarkware.com/software/JDepend.html>

There is another useful tool that is not covered in this workshop. Namely statical code analysis. It can be used to find bugs in software before they show up in testing or at runtime. FindBugs is such a tool that can be used together with Eclipse. It is strongly recommended to use this for the project in this course.

<http://findbugs.sourceforge.net/manual/eclipse.html>