The essential Java test toolkit

Table of contents

Copyright notice

About the author

(include picture)

I'm an independent test automation consultant living in The Netherlands. I have been in the field since 2006 and since then, I've worked with numerous test tools in various projects, both large and small. I especially like working with Java-based open source testing tools and libraries, which is why they are the subject of this ebook.

You can contact me at bas@ontestautomation.com or through my LinkedIn profile.

Introduction

During my career as a test automation consultant, I have come across a wide variety of tools and gadgets that make development and execution of automated tests a lot easier. Some of these tools are commercial tools, but a surprisingly large amount of them are open source and free for everybody to use to their liking.

In this ebook, I'll introduce some of the tools I have found most useful, complete with code examples. Where applicable, I will also show how these tools can be combined to create even more powerful automated test solutions.

Who should read this ebook?

Simply put: I've written this ebook for everybody with an interest in the implementation of useful open source test automation tools.

What's in this ebook?

This ebook contains:

- A selection of open source tools I have successfully used for automated testing
- Working examples of some of the most powerful features for each of these tools
- Download links for the Eclipse projects I have used to create the examples I use throughout this book

What's not in this ebook?

- Instructions on how to build a fully featured test automation framework. Each project is different, and therefore the requirements for an automated testing solution will vary depending on the project

you're working on. What works in one situation might not work as well in another. I leave it to you, the reader, to select and combine the tools you need and to determine the way you apply these tools in your project.

- Complete documentation and examples of all features for each and every tool. That's what the website of the tool is for. I'll make sure to include links to those sites, though.

Also, you may find your favourite tool is not featured at all. If you feel a particular tool is missing, please do send me an email at bas@ontestautomation.com and I'll gladly consider adding it in a future version of this ebook.

Any other suggestions, remarks or bribes can be sent to the above email address as well.

Jenkins and Ant

Before we dive into the test tools that form the main subject of this book, I would like to introduce another tool that is essential for successful automated testing projects: a continuous integration (CI) solution. This CI tool is going to be our basis for test execution and result reviewing in this book. Usually, in software development projects, the CI tool is also responsible for building and deploying the software to be tested. However, these steps in the CI process are beyond the scope of this book.

Personally, I have had great results using Jenkins. It's very easy to install and configure and is widely used, which means there's a vast amount of resources on the Internet that can help you out with any questions regarding Jenkins installation, configuration and general use.

Also, we are going to need a tool that contains the exact instructions to be executed when a Jenkins build is triggered. For this, I prefer to use Ant, the Apache solution for automation of software build processes. It´s also very easy to install, configure and use, just follow the instructions in the online manual.

I am very much aware that different projects and different persons prefer (or require) different CI and build solutions. The Jenkins + Ant route is only my personal preference and you're welcome to use your own alternatives.

JUnit

Website - http://junit.org/

What does the tool do?

JUnit is a unit testing framework for the Java programming language and is one of a family of unit testing frameworks collectively known as xUnit. JUnit is commonly used by Java developers to write unit tests to determine the quality of their code.

Installation and configuration

Installing and configuring JUnit is easy. Just download the latest version from the website, add the relevant .jar files to your IDE project and you're good to go. JUnit also comes pre-installed with any recent version of Eclipse. In fact, when you create a new Java project in Eclipse you can choose directly to include JUnit in your project as well.

When you download the JUnit library from the website instead, please note that you also need to include the Hamcrest library mentioned there in your project, or otherwise JUnit will not work properly.

Writing a test

To write a JUnit test method, you need to annotate it using the org.junit.Test method. It is considered good practice to place your JUnit tests in a separate class used only for testing. You can place this class either in a separate package or even in a separate source folder (for instance, test instead of src) that contains the same package structure as the src folder.

A typical JUnit test method consists of one or more actions that prepare and execute the test, such as:

- Instantiation of the class under test
- Calling the class method under test

- Performing an assertion that compares the method result to a predefined expected result

For example, let's assume we have a class Calculator with a method add(int x, int y) and we want to verify whether that method works correctly. We could do this using the following JUnit test method:

```
@Test
public void testAddition() {

	Calculator calc = new Calculator();
	Assert.assertEquals(8, calc.add(3, 5));
}
```

As you can see, assertions in JUnit are performed using the `assertEquals()` method. This method takes two arguments, compares them and reports the result. The arguments provided can be integers, but also strings, booleans and other Java object types.

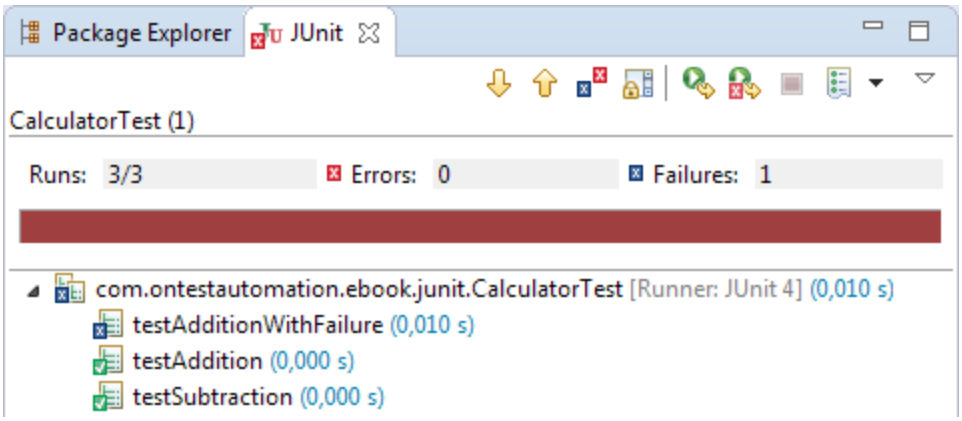
JUnit also contains a number of similar assertion methods, such as

- `assertTrue()` and `assertFalse()`
- `assertArrayEquals()`
- `assertNull()`

Running your test

To run your tests in Eclipse, simply right-click the class containing your JUnit tests and select '*Run As > JUnit Test*'. After test execution has finished, the test results appear in a separate tab.

In the example below we run a test class including the example test method presented above as well as two other test methods, one of which – `testAdditionWithFailure()` – deliberately results in a failure due to an incorrect expected value.



Useful features

Parameterized tests

To increase reusability and flexibility of your tests, JUnit supports parameterization of your test methods. To parameterize a JUnit test, you need to perform the following steps:

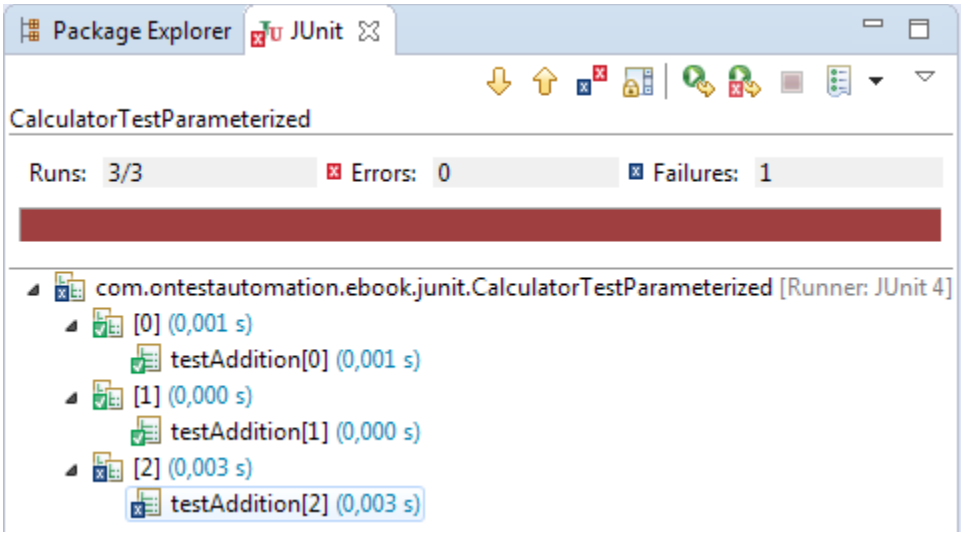
- Annotate your test class with `@RunWith(Parameterized.class)`
- Add (private) class variables that will contain the input and output parameters
- Add a constructor for your test class where the variables will be set
- Use the parameters in the test

For example, a parameterized test for the `add()` method of our Calculator class looks like this:

```
@RunWith(Parameterized.class)
public class CalculatorTestParameterized {

	private int x;
	private int y;
	private int sum;

	@Parameters
	public static Collection<Object[]> data() {

	    Object[][] data = new Object[][] { {2,3,5}, {4,4,8}, {5,8,14} };
	    return Arrays.asList(data);
	  }

	public CalculatorTestParameterized(int x, int y, int sum) {
```

```
            this.x = x;
            this.y = y;
            this.sum = sum;
        }

    @Test
    public void testAddition() {

            Calculator calc = new Calculator();
            Assert.assertEquals(sum, calc.add(x, y));
    }
}
```

If we run this test, the `testAddition()` method is run three times, once for each set of parameters in our parameter collection. The test results look like this:



The third test iteration fails due to an incorrect expected value for the addition (5 and 8 does not equal 14).

Rules

Rules can be used to create objects for configuring your test methods. For example, using a rule, you can specify the type of exception you expect when you perform a test:

```
public class CalculatorTestWithRule {

    @Rule
```

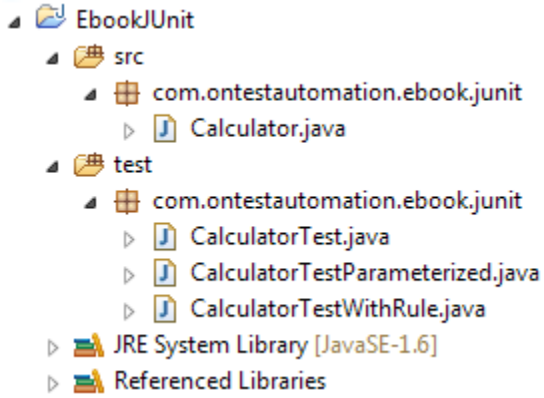
```
        // define a new ExpectedException
        public ExpectedException exception = ExpectedException.none();

        @Test
        public void testDivisionByZero() {

                // We are expecting an ArithmeticException when we divide by zero
                exception.expect(ArithmeticException.class);
                exception.expectMessage("/ by zero");

                Calculator calc = new Calculator();
                System.out.println(calc.divide(2, 0));
        }
}
```

When we run this test, it will pass as the implementation of the `add()` method of our Calculator class throws an `ArithmeticException` when we try to divide by zero, which is the expected behavior we specified.

Optimization and maintainability

Here are some more 3tips to optimize the use and increase the maintainability of your JUnit tests:

- JUnit tests should be placed in separate classes, so they don't get mixed up with actual production code. Personally, I like to place my JUnit test classes in a separate source folder called `test` (as opposed to `src` containing the source files to be tested). Then, I have the package structure in the `test` source folder mirror the package structure of the `src` source folder, for example:



- Only write JUnit tests for code that performs non-trivial or non-JVM-standard actions. For instance, there's no use in writing and maintaining tests for standard `get()` and `set()` methods for class variables, as you're pretty much only testing your JVM in that case.

- A JUnit test method should contain a single assertion. No more, no less. If your test contains more than one assertion, split it up into multiple test methods. This makes fault analysis in case one or more JUnit tests fail much easier as it minimizes the time required for pinpointing the correct piece of misbehaving code.

Continuous Integration

Integrating JUnit tests into our Continuous Integration framework is very straightforward.

Ant comes with a dedicated `junit` task to run JUnit tests as part of the build process. See for example the `build.xml` snippet below:

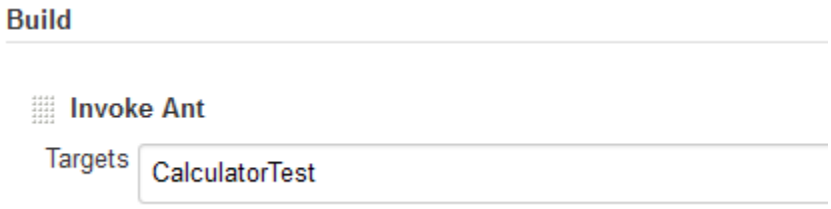
```
<target name="CalculatorTest">
    <mkdir dir="${junit.output.dir}"/>
    <junit fork="yes" printsummary="withOutAndErr">
        <formatter type="xml"/>
        <test name="com.ontestautomation.ebook.junit.CalculatorTest"
             todir="${junit.output.dir}"/>
        <classpath refid="EbookJUnit.classpath"/>
    </junit>
</target>
```

After these tests are run, you can use the `junitreport` task to gather the test results into a report:

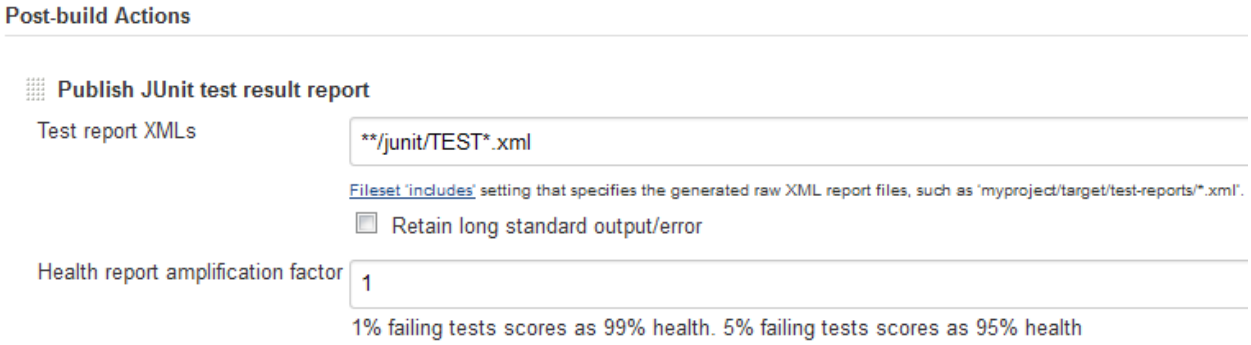
```
<target name="junitreport">
    <junitreport todir="${junit.output.dir}">
        <fileset dir="${junit.output.dir}">
            <include name="TEST-*.xml"/>
        </fileset>
        <report format="frames" todir="${junit.output.dir}"/>
    </junitreport>
</target>
```

Please note that both tasks can be automatically generated in Eclipse by exporting your project containing JUnit tests to an Ant build file.

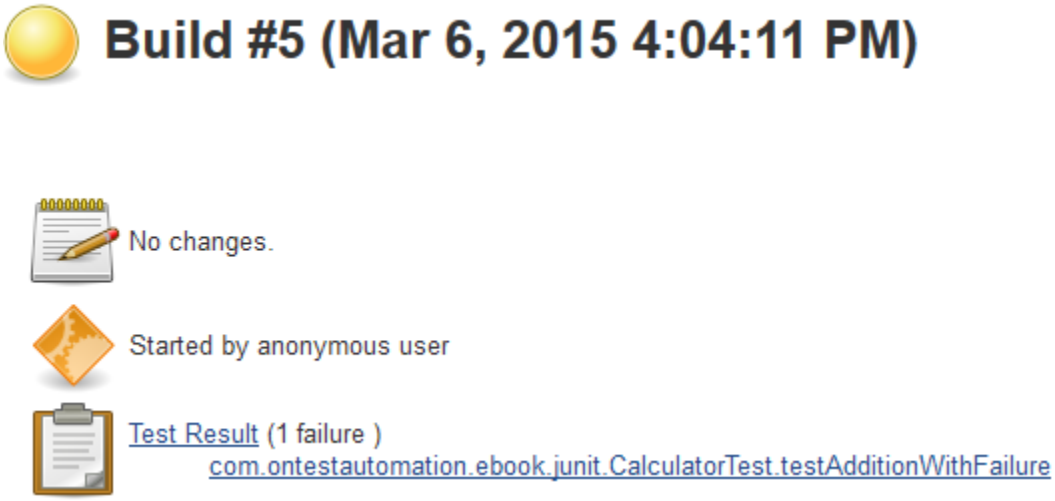
Next, we tell Jenkins to run this task as part of the build:



We also need to tell Jenkins where to pick up the test results:



When we start a new build, our JUnit tests are executed and the test results are integrated in the overall result of our build:



Finally, we can also view the console output for this build for additional information:


```
Started by user anonymous
Building in workspace C:\Program Files (x86)\Jenkins\workspace\EbookJUnit
[EbookJUnit] $ cmd.exe /C '"ant.bat CalculatorTest && exit %%ERRORLEVEL%%"'
Buildfile: C:\Program Files (x86)\Jenkins\workspace\EbookJUnit\build.xml

CalculatorTest:
    [junit] Running com.ontestautomation.ebook.junit.CalculatorTest
    [junit] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0,042 sec
    [junit] Test com.ontestautomation.ebook.junit.CalculatorTest FAILED

BUILD SUCCESSFUL
Total time: 0 seconds
Recording test results
Build step 'Publiceer rapport van de JUnit-testresultaten' changed build result to UNSTABLE
Finished: UNSTABLE
```

Downloads

A complete Eclipse project containing all of the code discussed in this chapter, as well as an example build.xml for continuous integration purposes, can be downloaded from here (LINK TO BE ADDED).