Lunds tekniska högskola
Datavetenskap, Nov 25, 2010
Görel Hedin
EDA260 Programvaruutveckling i grupp—projekt
Labb 3 (Test First): Bakgrundsmaterial

# JUnit in EDA260

## 1 Introduction

The JUnit framework is widely used for unit testing of Java code. It is developed by Kent Beck and Erich Gamma, and the ideas originate from a similar framework developed by Kent Beck for Smalltalk (SUnit). Today, there are similar frameworks for many other programming languages. This article gives a brief introduction to JUnit, and how to use it in the course EDA260—Programming in Teams.

JUnit supports unit testing, regression testing, and fixtures:

**Unit testing** The idea with unit testing is that for each class of production code, say `A`, you write a test class, say `TestA`, where you put methods that test the functionality of `A`. In each test method, you typically compute a value and compare it with the expected result.

**Regression testing** JUnit lets you run all your test methods in one go, so you can easily see which tests work and which fail. Typically, you use this to make sure that things that used to work don't stop working when you have made a change to the software. This is called *regression testing*, i.e., making sure that your software does not regress (go back to a less developed state).

**Fixtures** In a test, you need example objects or example object structures, to have something to run the test on. For example, if you are developing a class `Stack`, you want to build some example `Stack` objects to run your tests on. Such example objects or object structures are called *fixtures* in JUnit. Often, several test cases can use the same fixture, and it is useful to have a common method that defines how to build it. There is special support in JUnit for this.

Eclipse has built-in support for JUnit. You can run all your tests by clicking on a button, If they all work, you will see a green bar in the user interface. If any of the tests fail, you will see a red bar. A JUnit slogan says *keep the bar green*. The idea is that the normal case should be that all your tests work. Then, when you implement a new piece of functionality, make sure that you quickly get back to the green bar. You can do this by working in *baby steps*: Break down your work in small pieces so that you can implement a little piece of functionality, together with new test cases for it. The pieces should be so small that you don't risk a long period with the red bar.

## 2 JUnit 4.3

This report focuses on JUnit version 4.3 which makes use of *Java annotations* to identify test methods. Java annotations is a way of adding structured comments to a program. The annotations don't change the meaning of the program. But other tools, like JUnit, can use the annotations to call the annotated code in a particular way, or to check properties of the annotated program. JUnit makes use of annotations to find all the test methods in your code and run them.

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

public class TestStack{

    @Test public void popShouldReturnLatestPushedElement() {
        Stack s = new Stack();
        s.push(314);
        s.push(17);
        s.push(42);
        assertEquals(42, s.pop());
    }
}
```

Figure 1: A simple JUnit test case

## 2.1 Unit tests

Suppose you are implementing a class `Stack` for stacking integer numbers. One important part of the stack's behavior is that a pop should return the latest pushed value. Fig. 1 shows how you can write a JUnit test method that tests this behavior.

There are a few things to note in this example.

**test annotations** The test method has an annotation `@Test`. Annotations are always placed right before the declaration they are annotating, in this case the `popShouldReturnLatestPushedNumber` method. Because of this annotation, JUnit will know that this method is intended to be a test method, and can run it in a regression test. The annotation `Test` is declared in the package `org.junit`, which is why there is an import `org.junit.*` in the program.

**behavior-driven tests** The test method name `popShouldReturnLatestPushedNumber` is long, but it conveys the expected behavior that we are testing. This style of naming test methods is called *behavior-driven*: when you design test cases, you try to think about what behavior you would like the tested class to have, and name the test methods accordingly. This is nice because you can get an idea of the behavior of the class just by reading the names of the test methods.

**test method placement** The test method is placed inside a class, in this case `TestStack`. JUnit will create an instance of this class and call the test method of that instance. To make this work, both the test class and the test methods need to be `public`, and the test method should be `void` and have no arguments.

**assert methods** The `assertEquals` method is part of the JUnit framework. It has two arguments. The first is the *expected value*. The second is the *actual value*, computed using the `Stack` object. Make sure you write the arguments in the correct order: When a test fails, JUnit prints an error message based on these arguments. If you write them in the right order, the message will make better sense.

**the `Assert` class** The `assertEquals` method is a static method in the class `org.junit.Assert`. To be able to refer to it directly, without having to write `Assert.assertEquals`, we have added the *static* import of `org.junit.Assert.*`. This also gives us access to lots of other assert methods.

**JUnit API** The API for JUnit is available at `http://junit.sourceforge.net/javadoc/`. For example, take a look at all the assert methods available in the class `Assert`. Note, however, that this API shows the latest version of JUnit. So some parts of this API can only be used when using later versions than JUnit 4.3. Se `http://www.junit.org` and `http://junit.sourceforge.net/README.html` for a summary of changes between different JUnit versions.

## 2.2 Fixtures

In the test case described above, we created an example stack with the numbers 314, 17, and 42 in it. As you implement more test cases, you might find that you have use for the same example object, or *fixture*, over and over again. To make your test cases simpler, you can write code creating your example objects once and for all, using a `Before` annotation, as shown in Fig.2.

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

public class TestStack{
    Stack exampleStack = new Stack();

    @Before public void setUp() {
        exampleStack.push(314);
        exampleStack.push(17);
        exampleStack.push(42);
    }

    @Test public void popShouldReturnLatestPushedElement() {
        assertEquals(42, exampleStack.pop());
    }

    @Test public void testDepth() {
        assertEquals("depth of non-empty stack failed", 3, exampleStack.depth());
        assertEquals("depth of empty stack failed", 0, new Stack().depth());
    }
}
```

Figure 2: Using JUnit fixtures

To understand how this works, we need to get an understanding of how the JUnit tool finds the test methods and runs them. Typically, you run the JUnit tool on a directory, such as a complete project or an individual package. JUnit will find all classes that are in this directory, and then all methods annotated with `Test` in those classes. For each such test method, say $m$, it will create an object of the method's class, say $C$. It will then run all "before" methods in $C$, and then the test method $m$. The following pseudo code illustrates the process:

```
RUN TESTS FOR A DIRECTORY d:
  for each class C in d
      for each method m that is annotated with Test in C
          let aC = new C() in
              for each method b that is annotated with Before in C
                  aC.b();
              aC.m();
```

In particular, notice that each test method will run in its *own* instance of the test class. This way, each test method will get its own fresh objects, initialized by the "before" methods. This way, tests become independent: changes done by one test method, cannot affect any other test method. This is good. We don't want tests to depend on each other.

Things to notice in the Fig. 2.

**Before annotation** The example stack is now an instance variable in the `TestStack` object, instead of a local variable in the test method. The set up of the example object is done in a method `setUp` that is annotated by `Before`. Because of this annotation, the `setUp` method will be run before the test method is run.

**Independent test objects** As discussed above, it is important to note that JUnit runs each test method within its own test object: It will create one `TestStack` object for running the test method `popShouldReturn...`, and another `TestStack` object for running `testDepth`. This way, the two test methods will be *independent*: they will each get their own `exampleStack` object, and the execution of one test method will not affect the outcome of the other test method. So the fact that `popShouldReturn...` changes the `exampleStack` object does not affect the outcome of the `testDepth` method.

**Classic-style test names** The new test method `testDepth` is not named in the behavior-driven way, but the name here simply says something about *what* we are testing. It does not say anything about the expected behavior. We can call this the *classic style* of naming test methods. This is the naming style used originally in JUnit, before the idea of behavior-driven naming came up. It is useful to know about both these styles.

**Assert messages** The `testDepth` method calls `assertEquals` with an extra first argument, an *assert message*. This message is printed if the assert fails. All assert methods come in variants that take this extra argument. When you use the classic-style test names, it is recommended to use such assert messages to make it easier to understand the failures. When you use the behavior-driven names, there is often sufficient information in the test method name itself.

**Multiple asserts** The `testDepth` method contains two assert statements. The programmer found this convenient in this case, rather than having one test method for each assert. However, in general it is a good idea to not have very many asserts in the same test method. The reason is that if one assert fails, the rest of the asserts in that method will not be run. Your asserts become independent of each other if you put them in different test methods.

## 2.3 Testing exceptions

Sometimes your production code should throw an exception. How can you test that? Fig. 3 shows how to test that `pop` throws an exception if it is called on an empty stack.

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

class TestStack{
    ...
    @Test(expected = EmptyStackException.class)
    public void popShouldThrowExceptionOnEmptyStack()
    throws EmptyStackException {
        new Stack().pop();
    }
}
```

Figure 3: Testing an expected exception

The `Test` annotation here has a parameter `expected`. If the test method throws an exception object of the class `EmptyStackException`, the test will succeed. If it does not throw any exception, or throws an exception of some other class, the test will fail.

## 2.4 Mock objects

The `Stack` is a very simple object. What about when you have several objects in your production code that depend on each other? In unit testing, it is a goal to test each object in *isolation*. You don't want a test for one object to fail because it depends on another object that is incorrectly implemented. Also, when you are working in a team, you would like to develop different parts of the software in parallel, even if one part depends on another. A third problem is how to test objects that interact with other objects whose behavior might change over time. For example, the other object might be getting values from a database, whose contents change over time.

These problems can be solved with *mock objects*. If you are implementing a class `A` that depends on an object of class `B`, you simply replace the real `B` object by a mock object for `B`. This is an object that has just enough implementation to make the test work. It is free to fake the `B` implemention. For example, if `A` requires a result from calling a method in `B`, the mock `B` can return a result that works for that test, for example a constant `42` (rather than doing a real computation to return `42`). By using mock objects for `B` you can test `A`'s behavior in isolation.

Let's take a simple example. Suppose you are implementing a `BankAccount` class that should compute a monthly interest, based on the current interest rate for the account. The `BankAccount` calls a method in an `InterestRate` object, to get this rate. The real `InterestRate` object will return different interest rate values at different points in time. But we want to test the `BankAccount`, given some typical interest rates. The solution is to let `BankAccount` interact with the interface `AbstractInterestRate`. For the test case we will construct special `MockInterestRate` classes that implement this interface, each returning a value suitable for a particular test. In our production code, `BankAccount` will of course be connected to the real `InterestRate` object. Figures 4 and 5 illustrate the use of mock objects.

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

public class TestBankAccount{
    @Test public void testInterestRate() {
        BankAccount acc = new BankAccount(new MockInterestRate());
        acc.deposit(25000);
        acc.addMonthlyInterest();
        assertEquals(25000+25000*0.039/12, acc.balance(), 0.01);
    }
    class MockInterestRate implements AbstractInterestRate{
        public double currentInterestRate() { return 0.039; }
    }
}
```

Figure 4: Using a mock object in a test

```
public class BankAccount{
    public BankAccount(AbstractInterestRate interestRate) { ... }
    public void deposit(double amount) { ... }
    public double balance() { ... }
    public void addMonthlyInterest() { ... }
}
```

Figure 5: The `BankAccount` class

# 3 JUnit 3.8

There is an older version of JUnit that is very widely spread and used: JUnit 3.8. JUnit is backwards compatible, so test cases written using 3.8 will work also in the newer versions. If you work with older code, it is likely that you will see these older style test cases, so it is useful to know what they look like.

JUnit 3.8 was developed for Java 1.4 which did not support annotations. Instead of annotations, JUnit 3.8 uses subclassing and method naming conventions. Fig. 6 shows an example.

```
import junit.framework.*;

public class TestBankAccount extends TestCase{

    public void setUp() { ... }
    public void testSomething() {
        assertEquals(...);
    }
}
```

Figure 6: Tests written in the older JUnit version 3.8

Things to note:

- A class containing test methods is made a subclass of `TestCase` which is a class in the `org.junit` framework.
- A method setting up a fixture needs to be called `setUp`, overriding a method in `TestCase`. No `@Before` annotation is used.
- A test method needs to have a name starting with `test`, like `testSomething`. The JUnit framework recognizes that it is a test method because it is inside a subclass to `TestCase` and because it starts with `test`. No `@Test` annotation is used.

# 4 Learn more

While we have covered the essential parts of JUnit in this article, there is more to JUnit and unit testing in general that you might be interested in exploring on you own, later on. For example:

- In addition to the `@Before` annotation, there is also an `@After` annotation that you can use to annotate methods to run after each test method. This is useful if there is something that needs to be cleaned up after a test, such as closing a file.
- JUnit 4.3 also includes something called *theories*, that supports expressing general rules that cover a whole series of test cases.

- Later versions of JUnit support something called *hamcrest* notation, that better supports BDD (Behavior-Driven Development) where the asserts can be written in a more readable way.
- There are other unit testing frameworks than JUnit. Some of them support writing test cases in a special domain-specific language, rather than directly in Java. This can make the tests more readable. Examples include *RSpec* and *Cucumber*.
- There are many frameworks that simplify programming with mock objects. Examples include *EasyMock* and *JMock*. Some people distinguish between different kinds of mock objects, and would have called the mock object in our example a *stub*.

## 4.1 Using JUnit in Eclipse

To use JUnit in a new Java project in Eclipse, you need to add JUnit to the build path as follows:

- Create a new Java project
- Select the new project and open its properties: **context menu → Properties**
- Select **Java Build Path**
- Click on **Add Library...**
- Select **JUnit**
- Select JUnit library version **JUnit 4**
- Click **Finish**

Which version of JUnit 4 you get (e.g., 4.3, 4.4, etc.) will depend on the Eclipse version you are using.