Software-Architecture — Unit Testing (JUnit)

Prof. Dr. Axel Böttcher

3. Oktober 2011
Objectives

- Understand the concept of unit testing
- Know how to write unit tests
- Know when to write unit tests
Why Unit-Testing?

*Beware of bugs in the above code; I have only proved it correct, not tried it*

*D. E. Knuth*

- The idea of Unit-Testing is to test small units of code, i.e. single methods
- Unit-Testing is located between debugging and classical integration tests and acceptance tests (executed by QA-departments) and brings White-Box-Testing into development.
- Testability is a major quality criterion of a software architecture!
- Writing tests has positive influence on architecture and design of the software.

However, JUnit is used for any kind of tests today:

- Functional tests
- Integration tests
- Acceptance tests
- Stress tests
Tests are written parallel to the production code. Thus you avoid phases where the development team does nothing but writing tests (which might be regarded as unproductive by management)

Extrems case: Write a test before you write the program → **Test-Driven Development** (TDD). TDD is part of the idea of **Extreme Programming**.

**All Tests** are executed after every change of the code. Normally after each check in to the version control system – at latest within a nightly build.
JUnit is a Java tool that allows you to easily write tests.

It uses Annotations und Reflection (see one of the next chapters). During execution JUnit runs methods like:

```java
@Test public void ...()
```

JUnit offers assert-methods to formulate your test outcomes.

Example:

```java
assertEquals(5, model.getCurrentValue());
```

Here, the order is important: expected value, actual value, since an error report says: expected 5 but was ...
When to write a test?

- Basically you write a test class in parallel to every class
- For any non-trivial method (not for any simple getter or setter)
- Whenever an interface is unclear
- To isolate a bug
- Before refactoring of code that is yet untested (or undertested); especially when it is not your own code. The test is your protection against unwanted change of functionality or performance.
Carefully organize your tests

**Best practice:** Put tests into a separate directory tree with identical package names as production code (philosophy of Apache’s config tool *Maven*). Use `XyzTest` as name of the test for class `Xyz`.
Sketch of tests for classes that represent different calendars:

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

public class JulianDateTest {
    /* Test values taken from Dershowitz, Reingold: 
       Calendrical Calculations
       Cambridge University Press 1997 page 10 */

    @Test
    public void testStandard() {
        d = new JulianDate(10, 30, 1945);
        assertEquals(710347L, d.getFixed());
    }
}
```
Example (continued)

```java
@Test
public void testEaster() {
    JulianDate d = JulianDate.getEaster(2006);
    assertEquals(4, d.getMonth());
    assertEquals(16, d.getDay());
    assertEquals(2006, d.getYear());
}
```

Not very best practice, because the test stops as soon as the first `assertEquals` fails. But sometimes difficult to avoid..
Example (continued)

```java
@Test
public void testEaster() {
    JulianDate d = JulianDate.getEaster(2006);
    JulianDate easter =
        new JulianDate(4, 16, 2006);
    assertEquals(easter, d);
}
```

Good Practice. But depends on equals-method. Produces a readable error message as far as toString is reasonably implemented:

`java.lang.AssertionError: expected: <16.4.2006> but was: <15.4.2006>`
Different Asserts

- `assertEquals(expected, actual)`
- `assertEquals(expected, actual, delta)` for float and double obligatory; checks if $|expected - actual| < \delta$
- `assertNull`, `assertNotNull`
- `assertTrue`, `assertFalse`
- `assertSame`, `assertNotSame`

Whenever possible use this variant: `assertXXX( "Text describing the error", ...)`
Especially when testing for null or true/false this will produce a reasonable error message.
A Test case’s life

1. JUnit collects all `@Test`-Methoden in your test class via Reflection.
2. JUnit executes these methods in isolation from each other, and with undefined order.
3. JUnit creates a new instance of the test class for each test run in order to avoid side effects between tests.
4. Test run:
   4.1 An `@Before` annotated method is executed, if one exists.
   4.2 An `@Test`-method is executed.
   4.3 An `@After` annotated method is executed, if one exists.
5. This cycle repeats starting from step 3 until all test methods have been executed.
Integration in IDEs is very good: In Eclipse JUnit is a Launch Configuration on its own. Start a test case via context menu → Run As → JunitTest.
Advanced JUnit features

- Predefined maximum runtime of a test:
  ```java
  @Test(timeout = 1000l)
  ```
- Expected exception:
  ```java
  @Test(expected=NullPointerException.class)
  ```
- Flow of execution must not come to certain point: `fail("message")` makes the test fail anyhow
Parameterized tests

Idea: run one test with several parameter sets.

1. Annotate your test class with
   `@RunWith(Parameterized.class)`.

2. Implement a noarg public static method annotated with
   `@Parameters`, returning a `Collection` of Arrays.

3. Each element of the array must contain the expected value
   and all required parameters.

4. Implement a constructor setting these values to instance
   variables of the test.

5. Implement one test method using the parameters.
Example

```java
@RunWith(Parameterized.class)
public class GregorianEasterTest {
    private long year;
    private long day;
    private long month;

    @Parameters
    public static Collection<Object[]> getParameters() {
        return Arrays.asList(new Object[][] {
            {41, 161, 2006L},
            {41, 241, 2011L},
        });
    }

    public GregorianEasterTest(long month, long day, long year) {
        this.month = month;
        this.day = day;
        this.year = year;
    }

    @Test
    public void testEaster() {
        AbstractDate d = GregorianDate.getEaster(year);
        assertEquals(new GregorianDate(month, day, year), d);
    }
}
```
EMMA

EMMA is tool to check Code-Coverage of Unit-Tests. The corresponding Eclipse-Plugin shows:

1. Executed code
2. Not executed code
3. Partly Executed Code (in case of complex conditions)

```java
public static GregorianCalendar getEaster(long year){
  long century = year / 100 + 1;
  long shiftedEpact = (14 + 11 * (year % 19) - (3 * century) / 4 +
    ((5 + 8 * century) / 25)) % 30;
  long adjustedEpact = shiftedEpact;
  if((shiftedEpact == 0) || (shiftedEpact == 1 && 10 < (year % 19))){
    adjustedEpact++;
  }
  GregorianCalendar g = new GregorianCalendar(4, 19, year);
  return new GregorianCalendar(g.getFixed() - adjustedEpact, kDayAft);
}
```

Be careful: green does not mean that you have a sufficient amount of tests