Seminar 7 - JUnit Extensions, TDD

PV260 Software Quality

April 6, 2015

JUnit extensions

- JUnit is an extremely powerfull tool and virtually anything can be done using only the pure JUnit core functionality
- In some cases however we might benefit from using extensions of the basic functionality, syntactic sugar . . .
- These allow us to work faster, reduce the boilerplate code which brings no value, and making the test suite easier to maintain
- For most common needs both third party libraries and native JUnit extensions (some only in experimental branch) exist

JUnit extensions

- Property testing using randomized input
  - JUnit Theories `http://junit.org/apidocs/org/junit/experimental/theories/Theories.html`
  - junit-quickcheck `https://github.com/pholser/junit-quickcheck`
- Fluent API for assertions
  - Hamcrest (striped down version included in JUnit) `https://code.google.com/p/hamcrest/wiki/Tutorial`
  - FEST `https://github.com/alexruiz/fest-assert-2.x/wiki/One-minute-starting-guide`
- Parametrized /Data-Driven tests
  - JUnit Parametrized `http://junit.sourceforge.net/javadoc/org/junit/runners/Parameterized.html`
  - Zohhak runner `https://code.google.com/p/zohhak/`

Zohhak

https://code.google.com/p/zohhak/

Allows us to run one test on many sets of data, provided in annotation next to the testcase

```
@TestWith({
    "1,2,3",
    "-19,7,-12"
})
public void testAdd(int a, int b, int expected) {
    Calculator sut = new Calculator();
    int result = sut.add(a,b);
    assertEquals(expected, result);
}
```

Zohhak - Setup

To run the basic Zohhak example do the following:

- Download both zohhak jar and its dependency apache.commons-lang3 and place them on your test classpath
- Annotate the test class where you wish to use Zohhak with `@RunWith(ZohhakRunner.class)`
- Annotate the tests you wish to use zohhak with `@TestWith({...})`, this annotation will contain input data
- Run the test file as usual (Run Focused Test Method doesn't work for zohhak tests in NetBeans)

Zohhak - Data

- The Strings inside the `@TestWith({...})` each represent one test input
- Inside each of these input Strings individual arguments for the test are separated by commas (',')
- Types of the arguments are infered from the parameters of the test method and the arguments are coerced to these types before being passed to the test
    - Coercion of basic primitive types comes out-of-th-box
    - Custom coercion for any type can be written

Zohhak - Coercions

For more complex types we have to teach zohhak how to convert from String (the String in data annotation) to our type

```java
@Coercion
public Person toPerson(String input) {
    String[] split = input.split(";");
    Person person = new Person(split[0], split[1]);
    return person;
}
```

We can then use Person in our tests

```java
@TestWith({
    "John;Doe",
    "Frank;Perceval"
})
public void testWithPerson(Person person){
```

Test Coverage

In computer science, test coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite.

- High coverage does not necasarilly mean that your project has quality tests (there could be tests with no assertions, hardly maintainable tests . . .)
- However, low coverage can point to parts of insufficiently tested code which has a high chance of containing all kinds of bugs and other problems

Types of Coverage

Consider this code:

```
public int doIt(boolean c1, boolean c2, boolean c3) {
    int x = 0;
    if (c1)
        x++;
    if (c2)
        x--;
    if (c3)
        x+=3;
    return x;
}
```

Types of Coverage

- Statement coverage
  - Check that all statements in the code are executed
  - For 100% coverage single test input required *(true, true, true)*
- Branch coverage
  - Check that all possible results of conditions occur
  - For 100% coverage two test inputs required *(true, true, true)*, *(false, false, false)* or any other combination with both true and false for all conditionals
- Path coverage
  - Every possible path through the code is executed
  - For 100% coverage all possible combinations of inputs (and values for member attributes if there were any) must be used, thats 8 cases for this example

TDD - Overview

Test Driven Development: By Example, Kent Beck

> *Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.*

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication.
- Repeat . . .

Tennis Game Kata - Scoring

- Each player starts with 0 points
- The scoring then goes like this 0 $\rightarrow$ 15 $\rightarrow$ 30 $\rightarrow$ 40
- If `A` has 40 and scores, and `B` doesn't have 40, `A` wins
- If both have 40 and `A` scores, `A` has Advantage
- If `A` has Advantage and scores, they win
- If `A` has Advantage, `B` has 40 and scores, both are at 40 again
- Scores are written in the format 'A - B', e.g. '30 - 15'
- When `A` has Advantage, the score is written as 'A - 40'
- If scores are equal, e.g. both have 30, it is called '30 all'
- If both players have 40 points, it is called 'deuce'

Tennis Game Kata - Task

- Try to not skip ahead and always have passing tests for existing functionality before moving forward
- We want to create a TennisGame which has `scoredA()`, `scoredB()` and `showScore()`
- The show method should return score in format defined above, if there is a winner it gives 'winner: A/B'
- Also if there is a winner already and either `scoredA()` or `scoredB()` is called, exception should be thrown