

COPENHAGEN BUSINESS ACADEMY



Unit Testing - Recap

Hamcrest, Data-Driven Testing

Good Tests, Good Code

Unit Testing Learning Goals

- Explain about alternative matchers like **Hamcrest matchers** in a JUnit project? Provide examples to demonstrate, how to set up a project to use Hamcrest, the effect of using Hamcrest matchers compared to JUnit's basic Asserts.
- Explain the topic “**data driven testing**” backed up with real examples, using both “plain” JUnit and alternative libraries which lets you read test data from files (cvs, Excel, etc.)
- Explain properties of a **Good Unit Test**
- Explain patterns to write **Testable code**
- How, or if, **code coverage tools** like jacoco or similar can be used to measure the "quality" of our tests
- What is meant by a **test fixture**?
- Explain the rationale behind the ***Arrange - Act - Assert*** strategy for testing

(I believe topics in grey have already been covered well enough ☺)

Today's Recap

- Assert library Hamcrest
 - Write JUnit tests using Hamcrest Matchers
- Parameterized tests to write a data driven test
 - JUnit4, JUnit5
- Discuss patterns to write testable code

JUnit - a Quick Recap

- JUnit is a unit testing framework for the Java Programming Language, it provides, out of the box, the following features:
 - [Fixtures](#)
 - Test Suites
 - Test Runners (many IDE's have their own graphical runners)
 - Annotations
 - Assertions
- This is cool, because it's very easy to get started, but it also means that you are (initially) forced to use whatever you get.

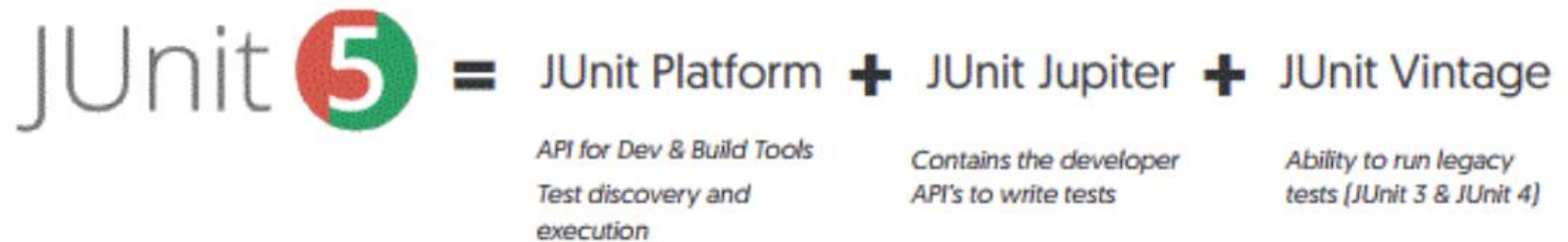
JUnit 4 - a Quick Recap

- Other frameworks like for example **Mocha**, a popular JavaScript testing framework has chosen another strategy.
 - It only ships with the actually testing framework required to design and execute the tests.
 - As a user you pick the assertion library that fits your needs (and there are many to choose among)
- As it happens, you can do something similar with JUnit, and select alternative assertion libraries, [Hamcrest](#) being the most popular one.
- Modern JUnit deployments actually ship with a limited Hamcrest implementation inside the JUnit-jar (org.hamcrest.CoreMatchers)

JUnit 5

JUnit5 Architecture

JUnit5 is modular:



JUnit 5 features

- **Annotations:** has renamed and extended some of the annotations used in JUnit4.
- **Assertions:** adds a few to be used with Java 8 lambdas
- **Assumptions:** Assumptions are similar to assertions, except that assumptions must hold true or the test will be aborted.
- **Parameterized tests** with value sources, e.g.
 - **@ValueSource:** Defines an array of primitive types
 - **@EnumSource:** Uses an *Enum* as a parameter source
 - **@MethodSource:** *method* provides a stream of arguments
 - **@CsvSource** and **@CsvFileSource:** Use parameters defined in CSV format, either in *String* objects or read from a file
- **RepeatedTest:** a test method can be repeated for a configurable number of times

Hamcrest

Matchers



Hamcrest is a framework that assists writing software test. It supports creating customized **assertion matchers**, allowing rules to be defined **declaratively**.

But why would we use Hamcrest? What's the problem with **all** the asserts we get for free with JUnit?

One of the problems lies in the word **all**, and also that they are not very readable.

Something like [this](#) would be much more clear

```
org.junit.Assert.  
assertArrayEquals(String message, float[] expecteds, float[] actuals, float delta) void  
assertEquals(Object expected, Object actual) void  
assertEquals(Object[] expecteds, Object[] actuals) void  
assertEquals(double expected, double actual) void  
assertEquals(long expected, long actual) void  
assertEquals(String message, Object expected, Object actual) void  
assertEquals(String message, Object[] expecteds, Object[] actuals) void  
assertEquals(String message, double expected, double actual) void  
assertEquals(String message, long expected, long actual) void  
assertEquals(double expected, double actual, double delta) void  
assertEquals(float expected, float actual, float delta) void  
assertEquals(String message, double expected, double actual, double delta) void  
assertEquals(String message, float expected, float actual, float delta) void  
assertFalse(boolean condition) void  
assertFalse(String message, boolean condition) void  
assertNotEquals(Object unexpected, Object actual) void  
assertNotEquals(long unexpected, long actual) void
```


Hamcrest

Why use Hamcrest?

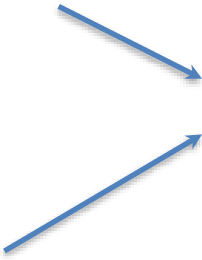
What does this test do?

```
@Test
public void t3(){
    assertTrue(aString.length() % 2 == 0 );
}
```

Let's see the error messages
(stack trace left out)

And this?

```
@Test
public void t4(){
    assertThat(aString.length(),isEven());
}
```



t3 Failed: java.lang.AssertionError
t4 Failed: Expected: an Even number but: was <5>, which is an Odd number

Which one do you prefer?

Which one do you prefer?

Hamcrest

Imperatively vs Declaratively

You might not think it's a big deal, but there is a great trend these days in the software community to favor Declarative strategies (describe what we want) in favor of imperative strategies (describe what to do (algorithm) to get what we want)

Imperative

```
assertTrue(aString.length() % 2 == 0 );
```

Assert that it is true, that the length of the string, modulus 2 is zero

Declarative:

```
assertThat(aString.length(), isEven());
```

Assert that the length of the string is even.

Hamcrest

Using Hamcrest



Let's see hamcrest in action 😊.

<https://github.com/Tine-m/hamcrestdemo>

Maven

Helps building project infrastructure in nice way 😊

Unit tests are usually included within the build process, which means they are run by a build tool like Maven.

Our focus right now is handling project dependencies via the **pom.xml**

JUnit 4.x Continued

Project Structure and Naming Conventions

Provides
JUnit 3
project

Using a typical Maven project (generated from a terminal as below)

```
mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart
```

We get a project layout as sketched below as our suggested Project Structure:

Suggested Project Structure

```
| pom.xml
|
|--src
|   +---main
|       |   \---java
|               \---demo
|                       App.java
|
|   \---test
|       |   \---java
|               \---demo
|                       AppTest.java
```

*Having our tests in a similar package as the class we test, allows the tests to access **Package Scoped Methods** (the default in Java)*

Naming Conventions

*Test Class Names should end with **Test***

*Name of methods should begin with **test (JUnit 3)** (and obviously the name should explain what the test does)*

What makes a Good Unit Test

Spend five min. with each of these links (only sections related to the header above):

- <https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>
- [https://msdn.microsoft.com/en-us/library/aa730844\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx)

Writing Testable Code

Effective Unit Testing [chap.7](#):

We will do this via a class discussion (and the exercises ;-)

- Follow the SOLID rules
- Reduce Dependencies
- Follow the [Single Responsibility Principle](#)
- Inversion of Control and Dependency Injections is our friend(s)
- Avoid hidden dependencies
- Use new with care
- Avoid logic in constructors
- Avoid Static Methods
- Avoid the Singleton
- Favor Generic Methods
- Create Simple Constructors
- Favor Composition over Inheritance
- Favor Polymorphism over Conditionals

When to Write Tests?

The best way to ensure testable code must be to write the tests first?!

You can write tests ...

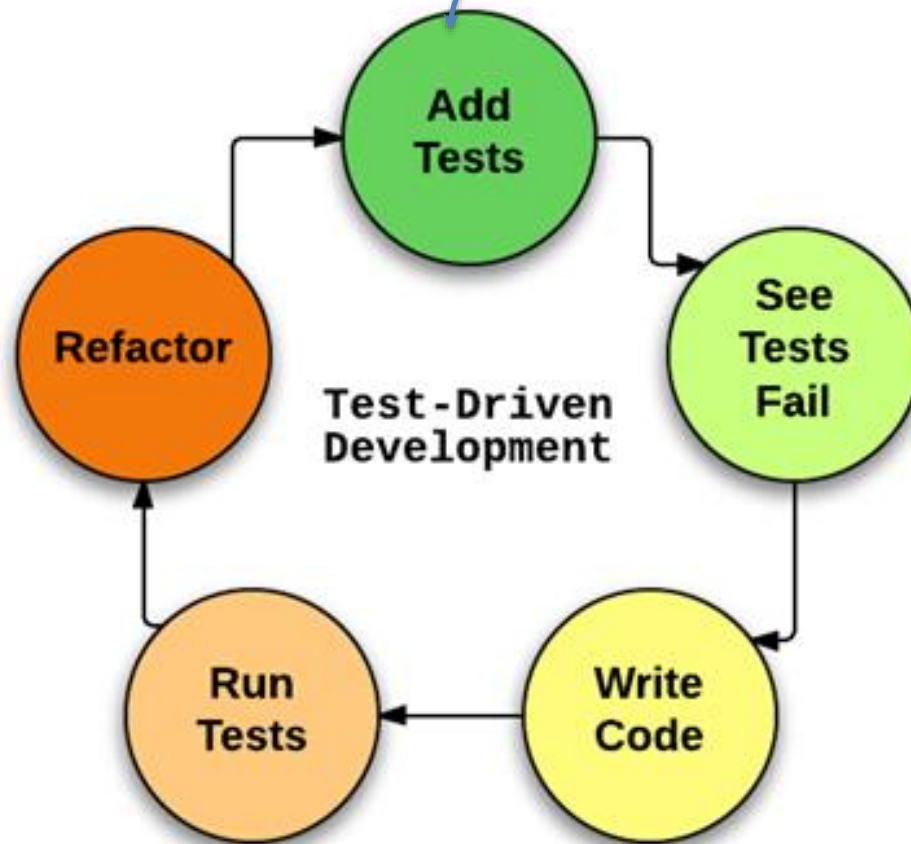
- After the code is written
 - ❖ If done *right* after, you will get instant feedback
 - ❖ Tests might be influenced by implementation
- While you write code
 - Co-design a unit and its tests together, in an iterative fashion.
 - Things gradually become more clear
- Before any code exists
 - ❖ This helps clarify the needed functionality (contract), i.e. becomes a design process
 - ❖ Test is not influenced by implementation knowledge

Test Driven Development TDD

We will use this document for our discussion:

<http://agiledata.org/essays/tdd.html>

Start before any code exists



Data-driven Unit Tests

What would you like to do for this specification?

- Write 15 tests, one for each "case" or
- Write 1 test, and run with 15 sets of inputs?

| ID | Test Case Description | Test Case Input | | | Expected Output |
|----|--|-----------------|------|----|-----------------|
| | | a | b | c | |
| 1 | Valid scalene triangle | 5 | 3 | 4 | Scalene |
| 2 | Valid isosceles triangle | 3 | 3 | 4 | Isosceles |
| 3 | Valid equilateral triangle | 3 | 3 | 3 | Equilateral |
| 4 | First permutation of two equal sides | 50 | 50 | 25 | Isosceles |
| 5 | Second permutation of two equal sides | 25 | 50 | 50 | Isosceles |
| 6 | Third permutation of two equal sides | 50 | 25 | 50 | Isosceles |
| 7 | One side zero length | 1000 | 1000 | 0 | Invalid |
| 8 | One side has negative length | 3 | 3 | -4 | Invalid |
| 9 | Three sides greater than zero, sum of two smallest is equal to the largest | 1 | 2 | 3 | Invalid |
| 10 | 2 nd permutation of 9 | 1 | 3 | 2 | Invalid |
| 11 | 3 rd permutation of 9 | 3 | 1 | 2 | Invalid |
| 12 | Three sides greater than zero, sum of two smallest is less than the largest? | 2 | 5 | 8 | Invalid |
| 13 | 2 nd permutation of 12 | 2 | 8 | 5 | Invalid |
| 14 | 3 rd permutation of 12 | 8 | 5 | 2 | Invalid |
| 15 | All sides zero | 0 | 0 | 0 | Invalid |

Data-driven testing (DDT) is a term used to describe testing done

- using a table of conditions directly as test inputs and verifiable outputs
- the process where test environment settings and control are not hard-coded.

In the simplest form, the tester supplies the inputs from a row in the table and expects the outputs which occur in the same row. The table typically contains values which correspond to boundary or partition input spaces.

Ref: https://en.wikipedia.org/wiki/Data-driven_testing

Data-driven Unit Tests

Using Parameterized Tests with plain JUnit4.x

JUnit 4.x has a (semi-cool ;-) feature called **parameterized tests**. This allow us to run the same test over and over again using different values.

A Parameterized Test requires you to perform these five steps:

- Annotate test class with `@RunWith(Parameterized.class)`
- Create a public static method annotated with `@Parameters` that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Create your test case(s) using the instance variables as the source of the test data.



Let's see an example:

<https://github.com/Tine-m/junitparameterizeddemo>

Data-driven Unit Tests

Using Parameterized Tests with plain JUnit4.x

How do you feel about what we just did. Was is cool, or would you have like to do it in another way?

What if you got specifications/test-results like this. What would you have to do (using a Parameterized test)?

Very often we have an issue like this: *We would like to read test-data tests from CSV, Excel or similar files.*

This is especially important if the data has been provided by someone else (QA, Clients, etc.)

| ID | Test Case Description | Test Case Input | | | Expected Output |
|----|--|-----------------|------|----|-----------------|
| | | a | b | c | |
| 1 | Valid scalenetriangle | 5 | 3 | 4 | Scalene |
| 2 | Valid isosceles triangle | 3 | 3 | 4 | Isosceles |
| 3 | Valid equilateral triangle | 3 | 3 | 3 | Equilateral |
| 4 | First permutation of two equal sides | 50 | 50 | 25 | Isosceles |
| 5 | Second permutation of two equal sides | 25 | 50 | 50 | Isosceles |
| 6 | Third permutation of two equal sides | 50 | 25 | 50 | Isosceles |
| 7 | One side zero length | 1000 | 1000 | 0 | Invalid |
| 8 | One side has negative length | 3 | 3 | -4 | Invalid |
| 9 | Three sides greater than zero, sum of two smallest is equal to the largest | 1 | 2 | 3 | Invalid |
| 10 | 2 nd permutation of 9 | 1 | 3 | 2 | Invalid |
| 11 | 3 rd permutation of 9 | 3 | 1 | 2 | Invalid |
| 12 | Three sides greater than zero, sum of two smallest is less than the largest? | 2 | 5 | 8 | Invalid |
| 13 | 2 nd permutation of 12 | 2 | 8 | 5 | Invalid |
| 14 | 3 rd permutation of 12 | 8 | 5 | 2 | Invalid |
| 15 | All sides zero | 0 | 0 | 0 | Invalid |

We could do this with plain JUnit, Parameterized Testing and a relevant file-library. There are however much better alternatives ;-)

JUnitParams – A Library to read test data from files

How to get started: With maven, it's extremely simple to get started

Add this to your pom.xml file

```
<dependency>
  <groupId>pl.pragmatists</groupId>
  <artifactId>JUnitParams</artifactId>
  <version>1.0.6</version>
  <scope>test</scope>
</dependency>
```

And the required imports to your test

```
import junitparams.FileParameters;
import junitparams.JUnitParamsRunner;
import junitparams.mappers.CsvWithHeaderMapper;
```

And you are ready to go;-)

With JUnitParams, all the clumsy
Parameterized-code from our first
example, can be re-written as
simple as this:

```
@Test
@FileParameters("src/test/resources/primes.csv")
public void testWithCSV(int num, String exp) {
    Boolean expect = Boolean.valueOf(exp);
    assertEquals(expect, primeNumberChecker.validate(num));
}
```

Example from https://www.tutorialspoint.com/parameterized_test.htm



Let's see an example:

<https://github.com/Tine-m/junitparamsdemo>

More info: <https://github.com/Pragmatists/JUnitParams/blob/master/src/test/java/junitparams/usage/SamplesOfUsageTest.java>

Data-driven Unit Tests

Triangle Program

The **JUnitParams - Library**, which can do a lot, but here, we will only use it to read test-data from a file.

Use the `JUnitParamsRunner.class` Runner

Specify a data-file, type, and that first line is a header (so skip it)

```
@RunWith(JUnitParamsRunner.class)
public class TriangleTest {
    ...
    @Test
    @FileParameters(value= "src/test/resources/triangle.csv",
                     mapper = CsvWithHeaderMapper.class)
    public void testWithCSV(String des, int a, int b, int c, String expectedResult)
    {
        assertEquals(des, expectedResult, triangleChecker(a, b, c));
    }
}
```

triangle.csv

| Description | a | b | c | Expected Result |
|--------------------------|---|---|---|-----------------|
| Valid scalene triangle | 5 | 3 | 4 | Scalene |
| Valid isosceles triangle | 3 | 3 | 3 | Isosceles |