TESTS

https://github.com/cnls/Tests

https://www.lynda.com/Developer-Programming-Foundations-tutorials/Welcome/124398/137955-4.html

http://junit.org/junit4/

http://junit.org/junit5/

https://github.com/howtoprogram/junit5-examples

http://dbunit.sourceforge.net/

http://www.guru99.com/selenium-tutorial.html

http://site.mockito.org/

https://dzone.com/articles/getting-started-mocking-java

https://www.eclemma.org/jacoco/

https://www.seleniumhq.org/download/

TESTING

- Aim of testing
- Difference between white box and black box testing
- Different types of tests
- Test driven development
- Implications of designing tests
- Determination of equivalence classes and boundary values
- Performing automated testing
- Code coverage

AIM OF TESTING

Software testing is an investigation conducted to provide information about the quality of the code and product giving an objective, independent view of the software.

Determine the correctness of software under the assumption of some specific hypotheses

Demonstrate behavior / Detect problems

Respond correctly / Find bugs / Satisfy requirements

Detach testing / Make repeatable

Inputs / Outputs / Expectations

BLACK BOX / WHITE BOX

White-box testing focuses on internal structures or workings

Internal structure/ design/ implementation is known to the tester

Black-box testing focuses on functionality

Internal structure/ design/ implementation is not known to the tester

MANY DIFFERENT TYPES OF TEST

UNIT TESTS

Smallest testable parts / Testing certain functions and areas

Verification, at the level of individual units and their methods and classes, that code works and continues to work as expected

A suite of tests can be run at any time during development to continually verify the code quality

INTEGRATION TESTS

Across layers

Integration tests are focused on groups of individually tested units and their collective interaction

Individual modules are combined and tested as a group after unit testing

Mockito

Maven dependency: org.mockito : mockito-all 1.10.19

DBUnit

Maven dependency: org.dbunit : dbunit 2.5.4

SYSTEM TESTS

Evaluate the whole integrated system and its compliance with specified requirements

Performed on the entire system in the context of the functional and non-functional requirements

Functional requirements: WHAT a software system should do

Non-functional requirements: HOW a software system should do something

Selenium

Maven dependency:

org.seleniumhq.selenium : selenium-java 3.11.0

org.seleniumhq.selenium : selenium-firefox-driver 3.11.0

org.seleniumhq.selenium : selenium-chrome-driver 3.11.0

Download browser driver

Chrome: http://chromedriver.storage.googleapis.com/index.html?path=2.36/

Firefox: https://github.com/mozilla/geckodriver/releases

USERINTERFACE TESTS

Identify the presence of defects in a product by using the graphical user interface

Does the user interface work correctly?

USER TESTS

Future users try to use a preliminary version of the system to see if they can solve their tasks

User performs a list of tasks while observers watch and take notes

ACCEPTANCE TESTS

Verifying a solution works for the user

A contract between the developers and the product owner on when a user story is implemented

TEST DRIVEN DEVELOPMENT

Writing tests before coding

Can be done in different programming languages

DESIGNING TESTS

What should be tested?

How to do testing?

Single purpose of test

Each test will reveal a potential error

Tests are independent of each other

Order of tests should not matter

Logic / Rules

Setters / Getters

To test it is needed to know what the expected behavior is:

- Return values
- Thrown exceptions
- Changes to objects and external components
- Calls to other objects

Enough input data sets should be used to make sure that:

- Methods have been called
- Both true and false branches have been executed in if statements
- Branches of switch statements have been executed
- Loops have been executed zero, one, and more times
- For every input data set, the expected output must also be specified

Equivalence classes

Group test case values into classes and select a value from each class, since all values in a class are expected to behave exactly the same way

Boundary values

Test the values on the edge of each test case value class, since it is expected that problems occur around edges

AUTOMATED TESTING

The use of special software, separate from the software being tested, to control the execution of tests and the comparison of actual outcomes with predicted outcomes.

Some software testing tasks can be laborious and time-consuming to do manually.

Test automation can automate repetitive tasks that would be difficult to do manually.

JUNIT

Simple framework to write repeatable tests

One of a family of unit testing frameworks collectively known as xUnit

JUnit API includes various classes and annotations to write test cases

Hamcrest

Matcher framework

Better readability and failure messages / Type safety / Flexibility / Custom matchers

Maven dependencies:

org.hamcrest : hamcrest-core 1.3

org.hamcrest : hamcrest-library 1.3

Junit4

Maven dependenciy: junit : junit 4.12

Junit5

Project properties – Sources – Source/Binary Format: 1.8

Maven dependencies:

org.junit.jupiter : junit-jupiter-engine 5.1.1

org.junit.vintage : junit-vintage-engine 5.1.1

org.junit.platform : junit-platform-runner 1.1.1

org.junit.platform : junit-platform-surefire-provider 1.1.1

Add new folder test/java in project/src folder

Add plugin

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.19.1</version>
  <dependencies>
    <dependency>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-surefire-provider</artifactId>
      <version>1.1.1</version>
    </dependency>
  </dependencies>
</plugin>
```

Naming convention

File names matter, since only Test*.java, *Test.java are automatically tested when building

Annotations / Assertions

Checked exceptions

CODE COVERAGE

Measures to which extent code has been tested

Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions

Testing cannot identify all the defects within software

Possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources

The development community is a bit divided on automated software testing

> Some people think you should have tests for 100% of all of your code, some believe that 80% is sufficient, some 50%, and some are content with 20%.

Coverage / Input / Combinations

Jacoco

Add plugin to plugins

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.1</version>
  <executions>
    <execution>
      <id>jacoco-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
      <configuration>
        <destFile>${project.build.directory}/coverage-reports/jacoco-ut.exec</destFile>
        <propertyName>surefireArgLine</propertyName>
      </configuration>
    </execution>
    <execution>
      <id>jacoco-report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
      <configuration>
        <dataFile>${project.build.directory}/coverage-reports/jacoco-ut.exec</dataFile>
        <outputDirectory>${project.reporting.outputDirectory}/jacoco-report</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Add argLine to properties

```
<argLine>${surefireArgLine}</argLine>
```

Report is generated in project/target/site/jacoco-report/index.html

Generate Jacoco report from project root folder with git bash…

```
mvn clean jacoco:prepare-agent install jacoco:report
```