## Midterm Assignment (20 Study points - mandatory)

The Midterm Assignment consists of two projects that each counts for 10 study points. Remember, this is a learning activity, so stop once in a while and think about what you are doing ☺

### Formalities

Hand-in on Moodle: Link to code on Github
Deadline: April 3rd at noon.

### Exercise/project 1

# Testing Real Life Code

*In this project we will use a relatively complex real life project, "JavaANPR - Automatic Number Plate Recognition System for Java" as the base for the exercise. So this exercise is not only a test-exercise, but also an exercise where you must prove (already having an AP degree in Computer Science ;-) that you can read and understand code.*

*The exercise is a study point exercise and will end up as <u>part of</u> an exam-question as sketched below.*

---

**Demonstrate your solution to the exercise "Testing Real Life Code".**

*You should (as a minimum):*

- *Explain the purpose of the Test (what the original test exposed, and what your test exposes)*
- *Explain about Parameterized Tests in JUnit and how you have used it in this exercise.*
- *Explain the topic Data Driven Testing, and why it often makes a lot of sense to read test data from a file.*
- *Your answers to the question; whether what you implemented was a Unit Test or a JUnit Test, the problems you might have discovered with the test and, your suggestions for ways this could have been fixed.*
- *The steps you took to include Hamcrest matchers in the project, and the difference they made for the test*

---

The project already provides a large number of tests, so we will dig into some of these and implement missing parts. We will use some of the more advanced features in JUnit, such as JUnit 4.x's - Parameterized Test and perhaps even Dynamic Tests available with JUnit 5.x ☺

**Getting started:**

- Clone the project from here: https://github.com/oskopek/javaanpr.git
- Open the project in your favourite IDE (it's a Maven project, so this should be straightforward)

1

**A)** Build and run the project, as explained in the project's readme-file. Select the menu item `Image->Load Snapshots` and load the provided snapshots (located in: src/test/resources/snapshots). You should observe that; when you use the "Recognize plate" button, not all plates are recognized correctly.

**B)** Localize the test file `net.sf.javaanpr.test.RecognitionIT` from within your IDE, find the two test methods and add something like this to the top of each:

```
logger.info("####### RUNNING: NAME_OF_TEST ######");
```

Execute the tests, either by:
- In NetBeans: Right click the file and do "Test File"
- From a terminal, do: `mvn surefire:test -Dtest=net.sf.javaanpr.test.RecognitionIT`

The tests will produce a large amount of console outputs, due to the many log statements in the code. If you execute the test via maven, make sure to find the two *logger.info* statements inserted above, in order to differentiate between outputs from the two tests.

**C)** Spend some time, in understanding the two test methods.
- The first will (successfully) test a single image. Try and replace the image with `test_074.jpg` to see the test fail
- The second test will read all snapshots (images), verify whether each one is recognized correctly, and update a count of `correctCount's` which it eventually will compare up against the expected value of correct snapshots.

The second test suffers from several problems.
1. First, it does not report failed/success for each image being tested. We must assume that the end goal for this project will be that all test passes, so it must be important to see exactly which images fail the test.
2. The second problem is that testing up against a count of `correctCount's` really isn't good. The risk is that one image that was expected to fail, will not fail, and another image that should pass, will not pass because of changes to the code. The existing test will not expose this scenario.

**D) Let's fix that ;-)** [1]

Your task is to create a new test class with a test that will report green/red for <u>each</u> image being tested as sketched using the NetBeans test-runner in this figure. You first impulse will probably be, to change the existing method `testSnapshots()` to use `asserts` instead of log-info's for failing tests. This however, will not work, since JUnit stops a test with the first failing assert, so this will only report the first failing image.



---
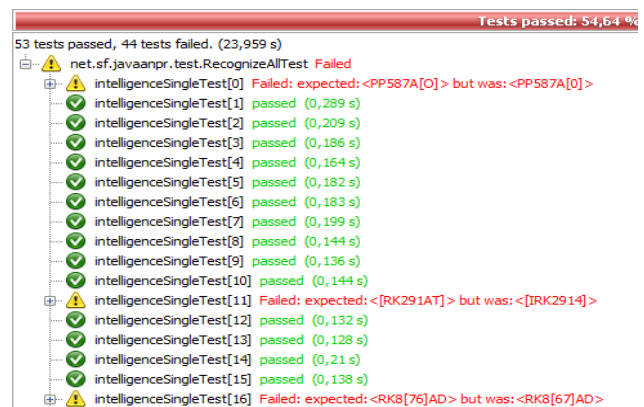
[1] This project is actually looking for contributors, so this could end up as an appreciated contribution to the project 🖥

Secondly you could create 96 different test's using the `intelligenceSingleTest()` method as a template ;-)

Obviously this is not an option, since it introduces a very bad smell in your code - which? Also, it will not work, out of the box, if new snapshots were added.

Luckily, we have learned about JUnit's Parameterized testing which can be used to solve the problem. Note: I don't suggest the JUnitParams library for this exercise, since it will require you to rewrite the existing file with expected results into a csv-file, or alternatively require that you write your own JUnitParams-mapper.

**Task:** Create a new test-class `RecognitionAllIT.java` in the `net.sf.javaanpr.test` package and solve the problem using a Parameterized Test.

**Hints:**
- Complete the first simple exercise with Parameterized Test given in the class, before you start.
- Make sure you understand the existing code. All necessary code is (almost) available in the two existing tests (how images are read, how expected results are read, how to call the recognize method etc.), you just need to put the pieces together the right way ;-)
- The way the expected results are read is actually pretty cool, and is definitely a strategy you can use for your future tests
- First give it a try, but if you find this too hard, the last page has a section "More Hints" ;-)

## Important before you start:

The project uses a maven plugin `checkstyle` to enforce a number of rules to be used when writing code for this project. You can disable this (unless you actually plan to contribute) with this statement in the `<properties>` section of your pom.xml file:

```
<checkstyle.skip>true</checkstyle.skip>
```

**E)**

We used JUnit for everything we did above. Was what you wrote a true Unit test? Or how would you classify this test?  Are there properties (things you have noticed) you could describe for this test (good, bad or just a property) and for the bad ones (if any), do you have suggestions to solve the problem.

**F)** This project does not use Hamcrest matchers for any of its test. Let's change that, starting with the class you wrote above. Add the necessary dependency to the pom-file, and change your test class to use hamcrest matchers to get cleaner test results.

**G)**

If you just can't get enough ;-) Fix the problem, one more time, this time using JUnit 5.x and dynamic tests.

**H) How to hand in**

Committing your own version of the full project would be silly for this small exercise. Create a new repository on github, and commit/push only your test class + a screenshot with the test results.

**Exercise/project 2**

# *Unit Testing, Testable Code, Mocking and Code Coverage*

*Since this is a test exercise, we do things in the opposite order of what is suggested in the class. That is, you are provided with un-tested code, that needs to be tested (and rewritten to make it testable), instead of writing test firsts.*

*Note: This is not a straightforward exercise. Expect to spend 2-4 hours on completing this exercise, especially since you need to stop, reflect, and learn from time to time.*

*It is a perfect exercise for pair programming, so that is what I suggest. If you don't have a Java background, team up with a Java programmer and enjoy, that when this top-up is over, you have an extra language to put on you CV :-)*

*The exercise has two major purposes.*

1. *Demonstrate ways to create testable code (in this case by refactoring untestable code into testable)*
2. *Use the Mockito framework to do state and behaviour based mock-testing*

*The exercise is a Study Point exercise and will end up as <u>part of</u> an exam question as sketched below:*

---

***Demonstrate your solution to the exercise "Unit Testing, Testable Code, Mocking and Code Coverage".***

*You should (as a minimum):*

- *Explain the necessary steps you did to make the code testable, and some of the patterns involved in this step*
- *Execute your test cases*
- *Explain basically about JUnit, Hamcrest, Mockito and Jacoco, and what problems they solve for testers*
- *Demonstrate how you used Mockito to mock away external Dependencies*
- *Demonstrate how/where you did state-based testing and how/where you did behaviour based testing*
- *Explain about Coverage Criterias, using the results presented by running Jacoco (or a similar tool) against you final test code.*
- *Explain/demonstrate what was required to make this project use, JUnit (Hamcrest), Mockito and Jacoco*

---

**Getting started:** Clone this project: https://github.com/Lars-m/startCodeForTesting1.git

Open the project in your favourite IDE, and execute the main method in the `JokeFetcher` class, either via the IDE, or maven (`mvn exec:java`) to see (important) code in action.

This project can fetch jokes from various open REST API's, and return the Jokes encapsulated in the `Jokes` class. This class also contain a Date String, adjusted to the caller's time zone.

The main class of interest is the `JokeFetcher` class and it's `getJokes(..)` method which you call like:

`getJokes("EduJoke,ChuckNorris,Moma,Tambal,ChuckNorris","Europe/Copenhagen")`

This will return a `Jokes` object containing five Joke instances, and the timestamp-string aligned to the time zone

passed in. The idea is, that this eventually will be used for our own REST API (not a part of this exercise), and users (from all over the world) can pass in a time-zone string with the request, to get the fetch time formatted with their local time. Se comments in code for more info.

**Exercise:**

**a)** Spend **at least** five minutes in familiarizing yourself with the code. It uses the `RestAssured` package to fetch jokes from the external API's, so this part, including JSON conversion, is simple (and <u>not</u> where you should place focus).

Your main task in this exercise is to test most classes and their public and package scoped methods (only test code that makes sense, so skip Joke, and Jokes - why?).

Since the code is definitely not designed for testing, you will have to rewrite it quite a bit, using some of rules introduced in "JUnit in Action".

**b)** Create the necessary test classes to test the code. I suggest you create a class to test the `DateFormatter` class and another to test `JokeFetcher` class (use the right naming strategy). Add a suite to test both.

*Important: Having done this you can complete this exercise in (at least) two ways. The hardcore/recommended (you are convinced about your OO and Mockito-skills) by following the suggested steps below (h1-h4), or you can follow the* **guided tour** *given on the following pages.*

## Your on your own track ;-)

**h1)** Locate class: `DateFormatter`, and create tests to verify the `getFormattedDate` method's public behaviour.

*Can you test this method (which of the rules for writing testable code does it breaks)? If not, refactor the code to something you can test.*

**h2)** Locate the class `JokeFetcher`   and write tests for the first two public methods which should be very simple.

**h3)** The next public method `getJokes(..)` is where  you find <u>most of the challenges</u> in this exercise.

*Remember that this is a unit test, so what is the first obvious problem is this method?*

The method breaks a number of the rules for writing testable code, such as, as a minimum:

- It violates the Single Responsibility Principle ([SRP](#)). The method has multiple responsibilities; it consumes information and also processes it
- It has hidden dependencies (relies on several external API's)
- it also does NOT favour polymorphism over conditionals
- It's basically impossible for a test, to mock away the external dependencies as it is

Refactor the code to use polymorphism (and perhaps a factory) and provide a single interface with a `getJoke()` method (please observe that the four getXX methods are all different, since jokes can, and is, returned in many different ways by the external API's).

Use `mockito` to mock away the external dependencies and test the getJokes(..) method and classes/methods which popped up during your refactoring.

**h4)** Add the necessary plugin to the pom.xml file to use the `jacoco` code coverage tool (or a similar one if you prefer) and measure the code coverage obtained by your tests. Reflect about the results obtained by running this tool

## Guided Tour for the rest of the exercise ;-)

*Note: Whenever you do "just follow on" exercises like this, the risk is that you will have more focus on just completing the exercise, instead of on learning. This last part is where you should place focus ;-)*

*Also, you are free to (and should) leave the guided tour whenever you like, if you feel you can complete it by yourself.*

**Testing the `DateFormatter` class:**

**1)** See if you can find the problem (seen from a testing perspective) with the `getFormattedDate(..)` method , fix it and test the method (with OK, and not OK inputs).

Otherwise read this [section](#)

Adding Date as an argument, as you hopefully did above, just moves the dependency problem up into the `JokeFetchers, getJoke(..)` method.

**2)** Create an *interface* `IDateFormatter`, from the `DateFormatter` class, and let the class implement this interface (you will probably notice a problem, related to the method being static which you will have to solve).

Write down as you go, all relevant observations you make. What can be a problem with static methods?

**3)** Change the `JokeFetcher` class using *Inversion of Control* and *Dependency Injection* to inject your refactored `DateFormatter` by: (Read about this [here](#)).

**4)** In the `JokeFetcher` 's main() method create an instance, inject it via the new constructor, and run main as a quick first test.

**Setup your project for Mockito:**

**5)** Verify that this dependency exists in the pom-file:

```
<dependency>

  <groupId>org.mockito</groupId>

  <artifactId>mockito-all</artifactId>

  <version>1.9.5</version>

  <scope>test</scope>

</dependency>
```

In the test class to test `JokeFetcher` class, add the @RunWith annotation on top of the class: `@RunWith(MockitoJUnitRunner.class)`

Create a (global) `IDateFormatter` Mock (using any of the possible ways to do this) and use this mock to

6

set up all your `JokeFetcher` instances.

Add these two imports to the top of your test class (to provide nice code completion):

```
import static org.mockito.Mockito.*;
```

```
import static org.hamcrest.CoreMatchers.*;
```

**Testing the `JokeFetcher` class**

As it is, the `JokeFetcher` is impossible to test (why), but we can write test for its simpler methods.

**6)** Add a test that verifies the `testGetAvailableTypes()` method (this method returns, until we change that, a hardcoded list from within the class which you need to use as the expected result).

*Hint: The **hamcrest** matcher `hasItems(..)` is perfect for this test, but you probably also want to verify the size of the list.*

**7)** Add a test to verify `isStringValid()`. If you can "*see*" this method from your test, explain why. If not, fix it using hints related to how to organize your test classes.

Verify that the method provides the expected result for *legal* and *not legal* inputs

**First simple test of the `getJokes(..)` method**

Now add a test method to verify (as much as we can) the functionality and "behavior" of the getJokes(..) method.

**8)** This method calls the provided `IDateFormatter` Mock created in step 5. Now we need to provide it with our expected value, so do that, for example like this (assuming your mock is named `dfMock`):

```
given(dfMock.getFormattedDate(eq("Europe/Copenhagen"), anyObject())).willReturn("17 feb. 2018
10:56 AM");
```

Call `getJokes(..)`, and verify that `getTimeZoneString()` returns the expected value.

*The test above did state based testing.*

**9)** Now verify that the `getJokes(..)` *behaves* as we expect. We do expect `getJokes(..)` to, underneath the surface, call the `getFormattedDate(..)` exactly one time. Verify this using Mockito's `verify(..)`

**Refactor the `JokeFetcher` Class and the `getJokes(..)` method**

The `JokeFetcher` class and `getJokes(..)` breaks a number of the rules for writing testable code, such as, as a minimum:

- Violates the Single Responsibility Principle (SRP). The method has multiple responsibilities; it consumes information and also processes it

- It has hidden dependencies (relies on several external API's)
- It does not use cool test patterns like dependency injection and interfaces
- it does NOT favor polymorphism over conditionals
- It's basically impossible for a test, to mock away the external dependencies as it is

We need to do a lot of refactoring, before we can test this method.

Before starting, you should take a quick look at this link and you should Google "*replace conditional with polymorphism*" and read up on this. You might also want to take a look at the Factory Method Pattern.

If the links above has given you some great ideas, just follow your "instincts". If you are still not quite sure how to continue, just follow on ;-)

**Refactoring to use polymorphism:**

**10)** Add a new package `testex.jokefetching` and in this package create the following interfaces/classes:

```
public interface IJokeFetcher {

  Joke getJoke();

}
```

This interface corresponds to the `Bird` class in the example given in the link above.

Create four classes that each should implement this interface:

`EduJoke, ChuckNorris, Moma and Tambal`

Move all code related to the individual class from the `JokeFetcher` class into these classes. As an example, this is how the `ChuckNorris` class should end.

```
public class ChuckNorris implements IJokeFetcher {

  @Override
  public Joke getJoke() {

    try{

    String joke = given().get("http://api.icndb.com/jokes/random").path("value.joke");

    return new Joke(joke,"http://api.icndb.com/");

    } catch(Exception e){

      return null;

    }
```

```
    }
}
```

Do not change anything in the code, what we do here is refactoring, so we should end with the same behaviour as when we started.

**Adding a Factory**

**11)** Add this interface (to the same package) as above

```java
public interface IFetcherFactory {

  List<String> getAvailableTypes();

  List<IJokeFetcher> getJokeFetchers(String jokesToFetch);

}
```

Add an implementation:

```java
public class FetcherFactory implements IFetcherFactory {

  private final List<String> availableTypes =
        Arrays.asList("EduJoke", "ChuckNorris", "Moma", "Tambal");

  @Override
  public List<String> getAvailableTypes(){ return availableTypes;}

  @Override
  public List<IJokeFetcher> getJokeFetchers(String jokesToFetch) {
    //This is for you to do, but wait
    return null;
  }
}
```

**Using the Factory**

**12)** Add a private `IFetcherFactory` field called `factory` to the `JokeFetcher` class, and change the constructor to allow us to inject an `IFetcherFactory` instance.

**13)** Remove, if not already done, all the `getXXXJoke()` methods which we have refactored into separate classes.

**14)** Change the `getJokes(..)` method into this:

```
public Jokes getJokes(String jokesToFetch, String timeZone) throws JokeException {

    checkIfValidToken(jokesToFetch);

    Jokes jokes = new Jokes();

    for (IJokeFetcher fetcher : factory.getJokeFetchers(jokesToFetch)) {

      jokes.addJoke(fetcher.getJoke());

    }

    String tzString = dateFormatter.getFormattedDate(timeZone, new Date());

    jokes.setTimeZoneString(tzString);

    return jokes;

  }
```

Make sure you code compiles (it won't <u>work</u> until you have completed the "*This is for you to do*" step in the `FetcherFactory,` but don't worry about that right now).

**Important:** If you, as most people, like to see "things" visualized, draw a quick (hand-written) UML diagram, representing the refactored architecture of your code.

When it compiles, say loudly, three times "How cool is polymorphism" ;-)

**Let's test and Mock**

**15)** Since we have added a new class (actually several new classes), we should test the `FetcherFactory` before we continue with testing the `JokeFetcher` class. If you have not yet completed the "this is for you to do.." section this is actually great, since it gives us a chance to prove that we can test the `JokeFetcher` class, even when all of its dependencies are not completed.

Create a Mock field for each of the four specialized classes `EduJoke,` `ChuckNorris,` `Moma` and `Tambal`

Set up the necessary code to let the **FetcherFactory** return a list with these four instances when called like this:

`List<IJokeFetcher> result = factory.getJokeFetchers("EduJoke,ChuckNorris,Moma,Tambal");`

Test and verify the public methods of the factory, especially that the method above returns an array with four objects, and that these objects match the expected `IJokeFetchers` types.

**Add the missing `JokeFetcher` tests**

Now we are ready to complete the tests for the `JokeFetcher` class. It is now refactored to allow us to inject both an `IDateFormatter` instance and an `IFetcherFactory` instance.

**16)** The factory needs to be able to return a number of `IJokeFetcher` instances via the call to `getJokeFetchers(..)`, so add this code at the start of your test (some of it might already be there):

```
private JokeFetcher  jokeFetcher;

@Mock IDateFormatter ifMock;

@Mock IFetcherFactory factory;

@Mock Moma moma;

@Mock ChuckNorris chuck;

@Mock EduJoke edu;

@Mock Tambal tambal;


@Before

public void setup() {

 List<IJokeFetcher> fetchers = Arrays.asList(edu, chuck,moma,tambal);

 when(factory.getJokeFetchers("EduJoke,ChuckNorris,Moma,Tambal")).thenReturn(fetchers);

 List<String> types = Arrays.asList("EduJoke","ChuckNorris","Moma","Tambal");

 when(factory.getAvailableTypes()).thenReturn(types);

 jokeFetcher = new JokeFetcher (ifMock, factory);

}
```

Before you continue, MAKE SURE you understand everything in the code above.

**17)** Now complete the all the tests.

For the test(s) of `getJokes(..)` you need to set up (in the GIVEN part of your tests) the specific (test) Joke, each of the four `IJokeFetcher` mocks will return

Verify that the method returns the expected values

Verify that the method (underneath the surface) acts as expected (`getFormattedDate(..)` is called exactly once and the factory's `getJokeFetchers(..)` the same.

**18)**

Add the necessary plugin to the pom.xml file to use the `jacoco` code coverage tool (or a similar one if you prefer) and measure the code coverage obtained by your tests.

Reflect about the results obtained by running this tool

**19)** Skip this step, and everything above could have been a waste of time ;-)

*Grab a well-earned cup of coffee or a beer ;-), and after that go back to page 1 and read the goals for the matching exam-question, and use that to reflect over what you have done.*