Mid terms assignment - test

Kasper Roland Pagh

Useful links:

All code: https://github.com/kasperpagh/MidTermAssignment

The licence plate test class:
https://github.com/kasperpagh/MidTermAssignment/blob/master/javaanpr/src/test/java/net/sf/javaanpr/test/RecognitionAllIt.java

The code for part two of the assignment can be found here:
https://github.com/kasperpagh/MidTermAssignment/tree/master/startCodeForTesting1

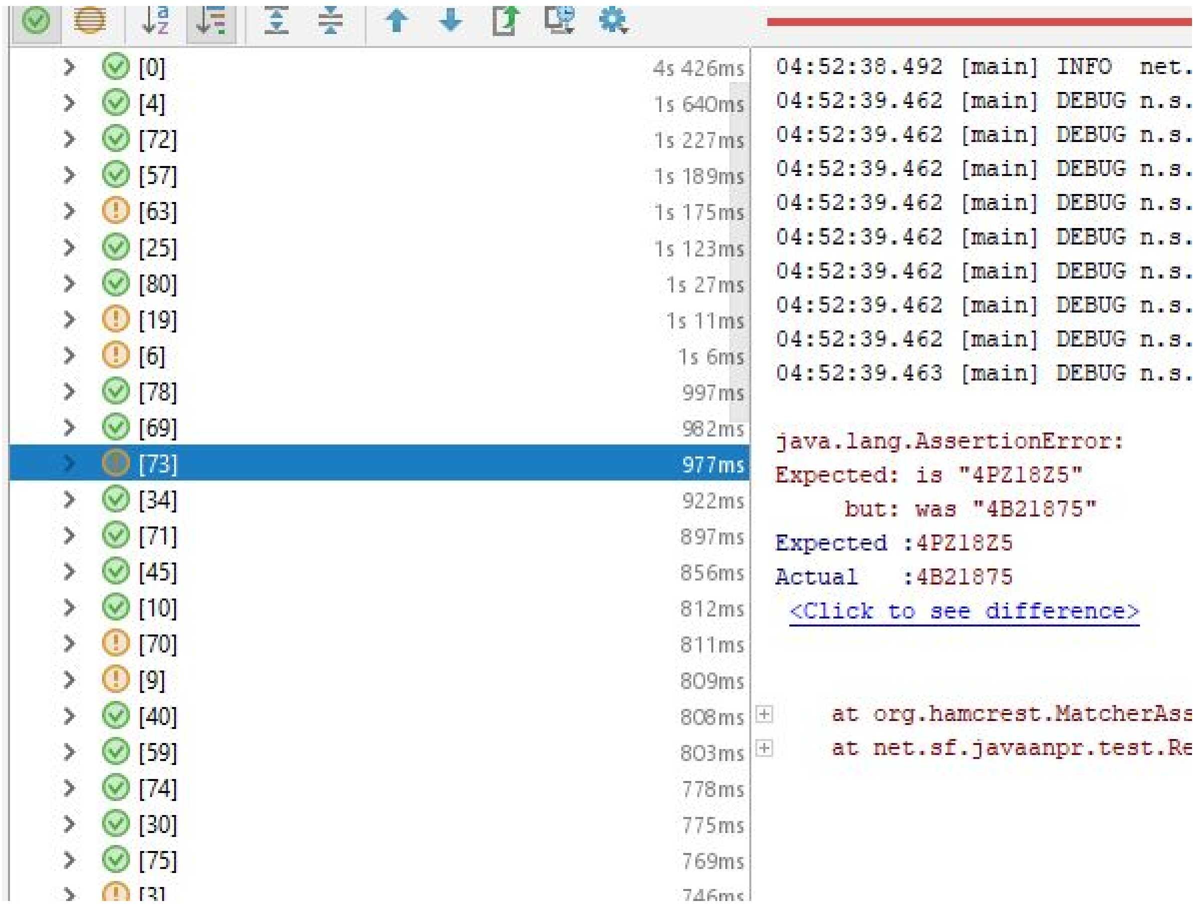
Monopoly game (as it is):
https://github.com/kasperpagh/MidTermAssignment/tree/master/MonopolyTDDGame

Assignment 1)

https://github.com/kasperpagh/MidTermAssignment/blob/master/javaanpr/src/test/java/net/sf/javaanpr/test/RecognitionAllIt.java

In its original implementation the test was quite bare bones, it knew that 44 out of the 97 provided images, but now which ones. Since all the checks ran in the same unit test, the output cluttered up the log to such a degree that actually reading the test's output became difficult.

To fix this issue we used a technique known as parameterized testing. This allowed us to run each of the nearly hundred test images, through it's own unit test - thus giving us a much more readable output



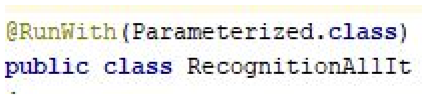
```
[0]   4s 426ms
[4]   1s 640ms
[72]  1s 227ms
[57]  1s 189ms
[63]  1s 175ms
[25]  1s 123ms
[80]  1s 27ms
[19]  1s 11ms
[6]   1s 6ms
[78]  997ms
[69]  982ms
[73]  977ms
[34]  922ms
[71]  897ms
[45]  856ms
[10]  812ms
[70]  811ms
[9]   809ms
[40]  808ms
[59]  803ms
[74]  778ms
[30]  775ms
[75]  769ms
[3]   746ms
```

```
04:52:38.492 [main] INFO  net.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.462 [main] DEBUG n.s.
04:52:39.463 [main] DEBUG n.s.

java.lang.AssertionError:
Expected: is "4PZ18Z5"
     but: was "4B21875"
Expected :4PZ18Z5
Actual   :4B21875
 <Click to see difference>


    at org.hamcrest.MatcherAss
    at net.sf.javaanpr.test.Re
```

ill. 1. Each of the test images are clearly listed, and you can click it for more info.

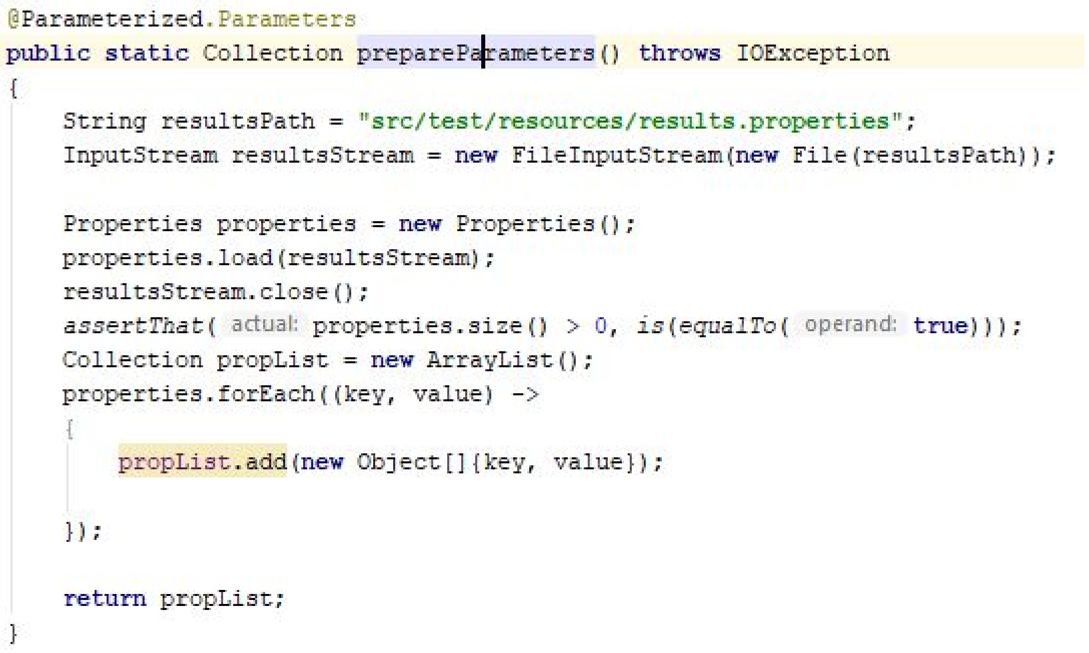
Creating parameterized tests in JUnit involves three steps.

1. Adding the annotation to the class.



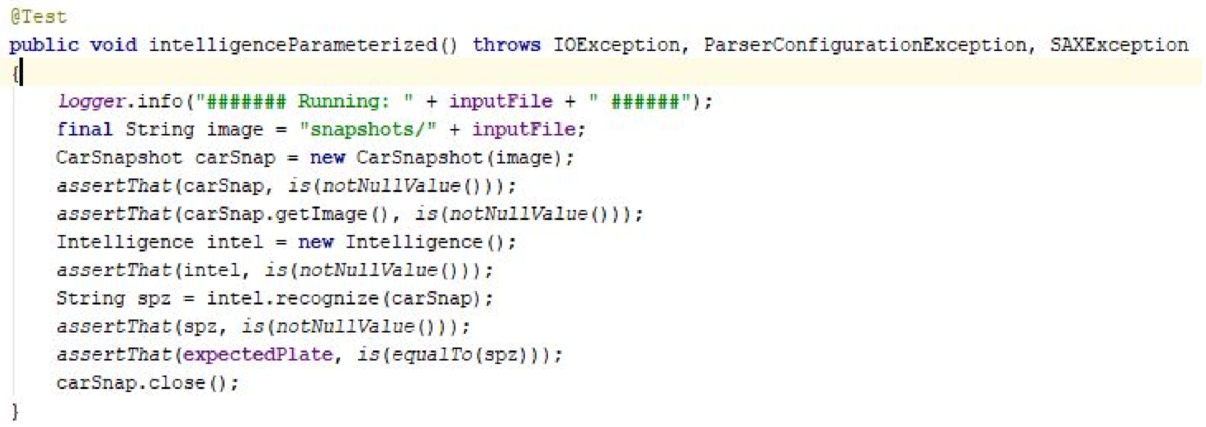
```
@RunWith(Parameterized.class)
public class RecognitionAllIt
```

2. Preparing the parameters - a list containing the test objects.



```
@Parameterized.Parameters
public static Collection prepareParameters() throws IOException
{
    String resultsPath = "src/test/resources/results.properties";
    InputStream resultsStream = new FileInputStream(new File(resultsPath));

    Properties properties = new Properties();
    properties.load(resultsStream);
    resultsStream.close();
    assertThat( actual: properties.size() > 0, is(equalTo( operand: true)));
    Collection propList = new ArrayList();
    properties.forEach((key, value) ->
    {
        propList.add(new Object[]{key, value});

    });

    return propList;
}
```

3. The last thing needed is the test itself - a test that must be run exactly once for every one of the tests parameters.



```
@Test
public void intelligenceParameterized() throws IOException, ParserConfigurationException, SAXException
{
    logger.info("####### Running: " + inputFile + " ######");
    final String image = "snapshots/" + inputFile;
    CarSnapshot carSnap = new CarSnapshot(image);
    assertThat(carSnap, is(notNullValue()));
    assertThat(carSnap.getImage(), is(notNullValue()));
    Intelligence intel = new Intelligence();
    assertThat(intel, is(notNullValue()));
    String spz = intel.recognize(carSnap);
    assertThat(spz, is(notNullValue()));
    assertThat(expectedPlate, is(equalTo(spz)));
    carSnap.close();
}
```

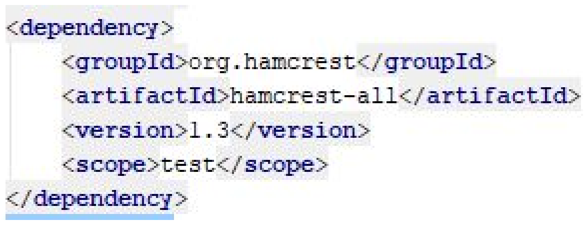
4.

This *RecognitionAllItis* test (from which the above screenshot comes from) is a good example of data driven testing. This design makes it easy to manage the test data, in regards to their “freshness”. It also allows the developer to have many different sets of data, such that testing the application in various different states is possible.

Data driven design also draws strength from the fact that having all your test data in a external file, allows you to quickly change the data whenever you want to.

A problem with the Recognition tests in general is that they are not very useful on their own. While the data source is external and the use of data driven design patterns ensures that consistent tests for the same builds, you would need some additional logic to ensure that the program still worked as expected. For example, some functionality that only activates if a test pass or fail deviates from the normal results (since the pictures have not changed, something in the code must be off).

When working with a maven project, adding external dependencies becomes quite easy and quick, to include the *Hamcrest* library to the project, was just a question of including the following snippet in the *pom.xml* file:



```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
```

ill. 2. With Hamcrest declared in the pom file maven takes care of the rest.

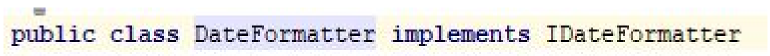
In this specific project we hamcrest was useful due to it’s more verbose failure messages, which comes in quite handy when writing test cases.

Assignment 2)

Data format

When I first got the code the dateformatter class relied heavily on an external API for supplying the current timezone, The problem is mainly the single call to *TimeZone.getTimeZone(timeZone)*, inside the *getFormattedDate* method of the class.

While this works great in practice, it is almost impossible to test since the state of the program is defined by outside influence. Therefore we need to refactor the method according to the rules of dependency injection and inversion on control, where we instead provide some instance of an interface as input, thus taking the control away from the *DateFormatter class.*

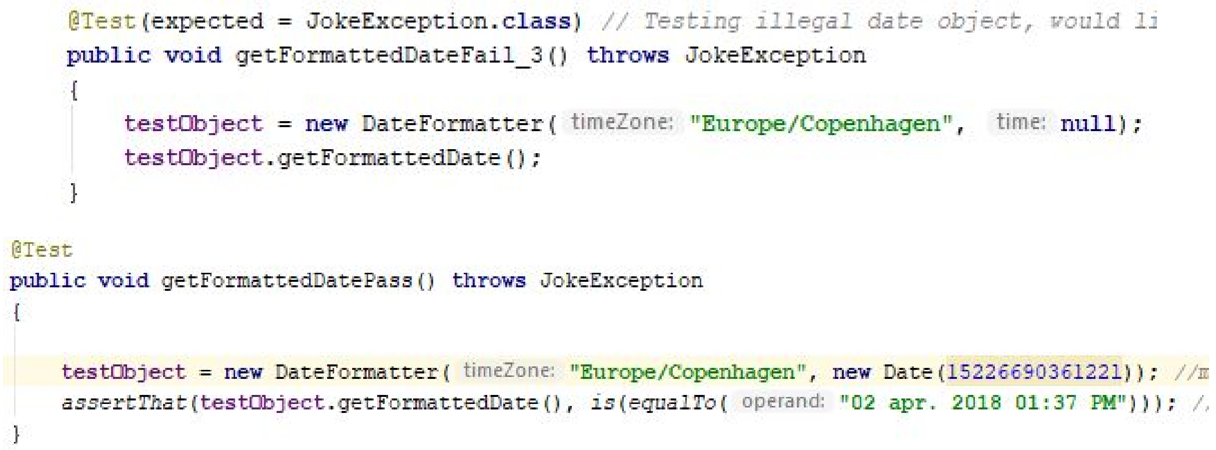


```java
public class DateFormatter implements IDateFormatter
```

ill. 3. The DateFormatter implements an interface making it much more flexible to test.

In the end i chose to refactor the DateFormatter class in such a way, that it takes two inputs upon construction - a String representing the timezone and an instance of a java.util.date object, defining the current time.

This means that i can call the method like this in my tests:



```java
    @Test(expected = JokeException.class) // Testing illegal date object, would li
    public void getFormattedDateFail_3() throws JokeException
    {
        testObject = new DateFormatter( timeZone: "Europe/Copenhagen",  time: null);
        testObject.getFormattedDate();
    }

@Test
public void getFormattedDatePass() throws JokeException
{

    testObject = new DateFormatter( timeZone: "Europe/Copenhagen", new Date(1522669036l221)); //m
    assertThat(testObject.getFormattedDate(), is(equalTo( operand: "02 apr. 2018 01:37 PM"))); /,
}
```

ill. 4. A couple of tests from the DateFormatterTest class. Notice how the second (bottom most) test is designed in such a way, that the output is always constant.

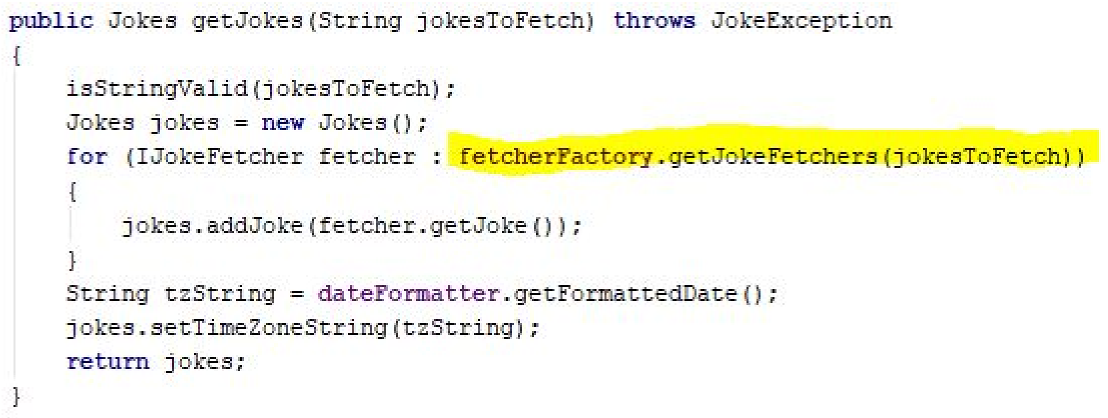
The *getFormattedDatePass* unit test (pictured above) is of special interest in regards to this change, since this new behavior of the class allows me to simply hardcode a time and timezone, such that the output of the unit tests is always assured.

Joke fetcher

The *JokeFetcher* class had similar problems to the dateformatter class. first of all; the class contained calls to four external REST APIs, which makes the state of the object hard to decide.

The class also had too many different responsibilities, ranging from gathering the data from the api, aggregating results, validating input, and sending the jokes to the user.

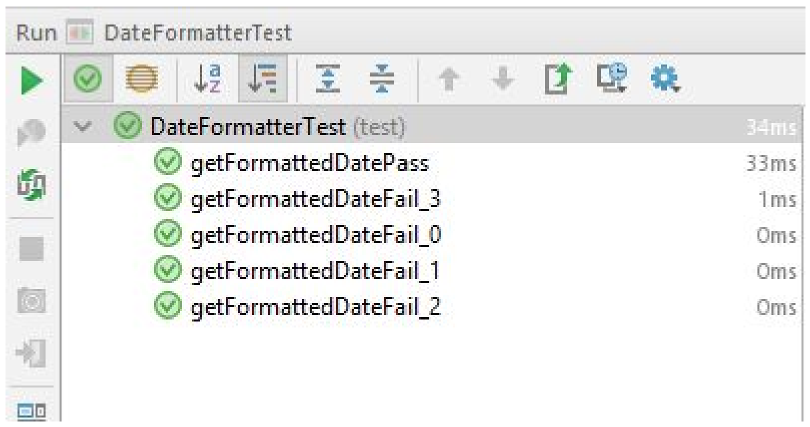
The solution, in the end, was to move all the logic associated with retrieving the jokes from the internet over to new classes, that all implemented the *IJokeFetcher* interface.



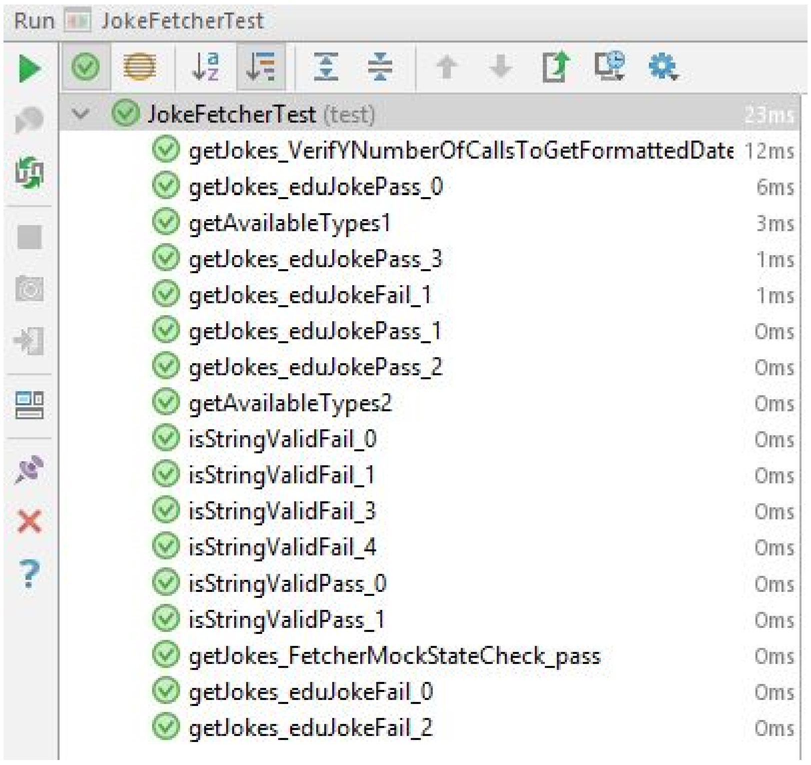
```
public Jokes getJokes(String jokesToFetch) throws JokeException
{
    isStringValid(jokesToFetch);
    Jokes jokes = new Jokes();
    for (IJokeFetcher fetcher : fetcherFactory.getJokeFetchers(jokesToFetch))
    {
        jokes.addJoke(fetcher.getJoke());
    }
    String tzString = dateFormatter.getFormattedDate();
    jokes.setTimeZoneString(tzString);
    return jokes;
}
```

ill. 5. Each of the test images are clearly listed, and you can click it for more info.

I decided to create a separate test class called *DataFormatterTest* and *JokeFetcherTest.* The test of the dateformatter is relatively simple, with just five unit tests. Four of these test test the behaviour of the dateformatter class if it's provided with illegal input, while the last one verifies the behaviour of the class, if provided with a very specific input (pictured above - ill. 2).



```
Run  DateFormatterTest

DateFormatterTest (test)          34ms
    getFormattedDatePass          33ms
    getFormattedDateFail_3         1ms
    getFormattedDateFail_0         0ms
    getFormattedDateFail_1         0ms
    getFormattedDateFail_2         0ms
```



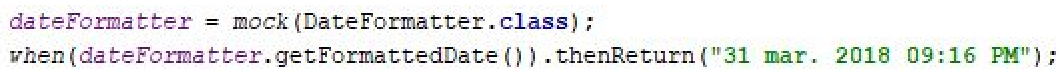
```
Run  JokeFetcherTest
JokeFetcherTest (test)                              23ms
    getJokes_VerifYNumberOfCallsToGetFormattedDate  12ms
    getJokes_eduJokePass_0                           6ms
    getAvailableTypes1                               3ms
    getJokes_eduJokePass_3                           1ms
    getJokes_eduJokeFail_1                           1ms
    getJokes_eduJokePass_1                           0ms
    getJokes_eduJokePass_2                           0ms
    getAvailableTypes2                               0ms
    isStringValidFail_0                              0ms
    isStringValidFail_1                              0ms
    isStringValidFail_3                              0ms
    isStringValidFail_4                              0ms
    isStringValidPass_0                              0ms
    isStringValidPass_1                              0ms
    getJokes_FetcherMockStateCheck_pass              0ms
    getJokes_eduJokeFail_0                           0ms
    getJokes_eduJokeFail_2                           0ms
```

ill. 6. Two screenshots showing the result of my two test classes.

All of my tests are JUnit tests (JUnit 4), while i used Hamcrest for assertions. This is because Hamcrest's *assertThat* provides a ton of functionality such as: type checks, improved readability, improved failure messages as well as a lot of flexibility.

Mockito

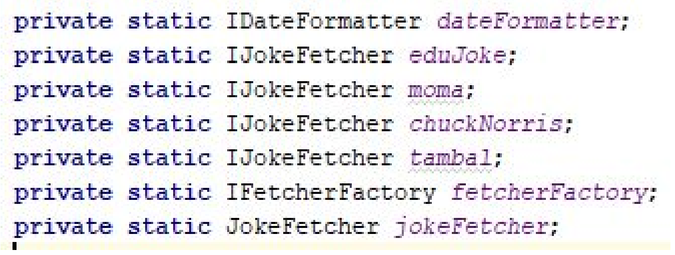
Mockito handles the job of creating the mock objects, that is heavily relied on by the *JokeFetcherTest.*



```java
dateFormatter = mock(DateFormatter.class);
when(dateFormatter.getFormattedDate()).thenReturn("31 mar. 2018 09:16 PM");
```

ill. 7. The use of Mockito to create a dateformatter before the class have even been made to work.

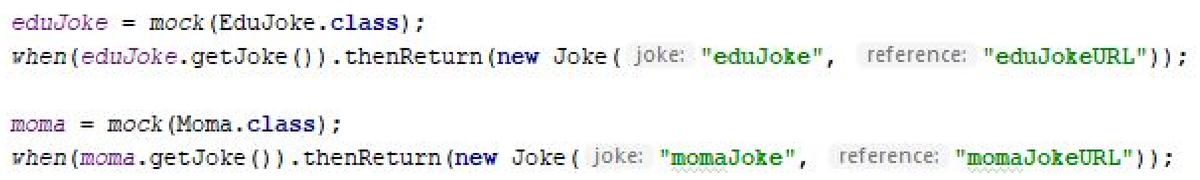
I primarily used Mockito in the *JokeFetcherTest* class.



```java
private static IDateFormatter dateFormatter;
private static IJokeFetcher eduJoke;
private static IJokeFetcher moma;
private static IJokeFetcher chuckNorris;
private static IJokeFetcher tambal;
private static IFetcherFactory fetcherFactory;
private static JokeFetcher jokeFetcher;
```

ill. 8. A screenshot of the declaration of the variables that i mock

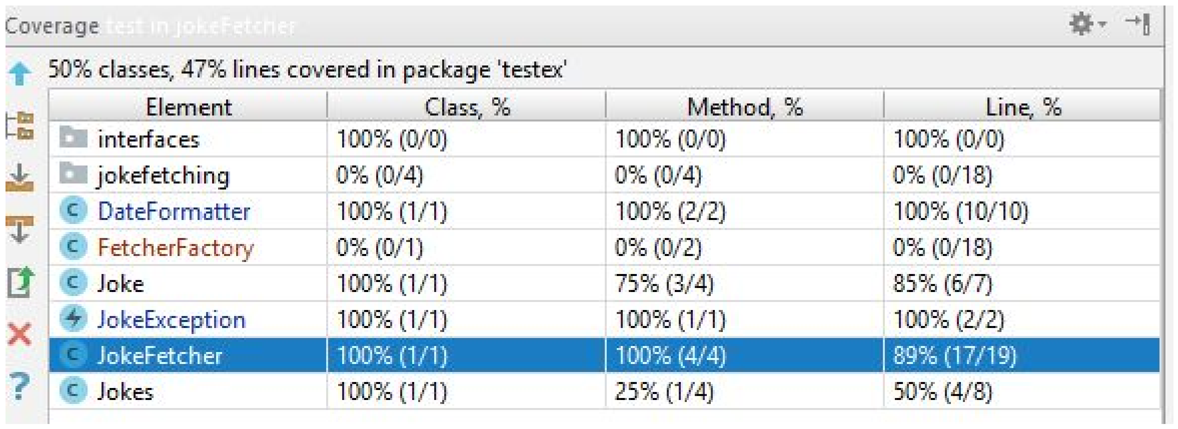
I first create the variables i want to mock (they are declared static, so that i can create the mocks in the @BeforeClass setup method), i then proceed to instantiate the variables as mocks, and give them appropriate return values, for different operations.



```
eduJoke = mock(EduJoke.class);
when(eduJoke.getJoke()).thenReturn(new Joke( joke: "eduJoke",  reference: "eduJokeURL"));

moma = mock(Moma.class);
when(moma.getJoke()).thenReturn(new Joke( joke: "momaJoke",  reference: "momaJokeURL"));
```

ill. 9. Making joke mocks return a predefined Joke.

For this project i decided not to use Jacoco, but instead used a plug-in for intelliJ that accomplishes much the same thing. Which is to say it gives you an ironclad metric for how much your tests actually cover.



Coverage test in jokeFetcher

50% classes, 47% lines covered in package 'testex'

Element	Class, %	Method, %	Line, %
interfaces	100% (0/0)	100% (0/0)	100% (0/0)
jokefetching	0% (0/4)	0% (0/4)	0% (0/18)
DateFormatter	100% (1/1)	100% (2/2)	100% (10/10)
FetcherFactory	0% (0/1)	0% (0/2)	0% (0/18)
Joke	100% (1/1)	75% (3/4)	85% (6/7)
JokeException	100% (1/1)	100% (1/1)	100% (2/2)
JokeFetcher	100% (1/1)	100% (4/4)	89% (17/19)
Jokes	100% (1/1)	25% (1/4)	50% (4/8)

ill. 10. A screenshot of the test coverage of the program. As you can see i placed almost all the effort in testing the DateFormatter- and JokeFetcher classes.