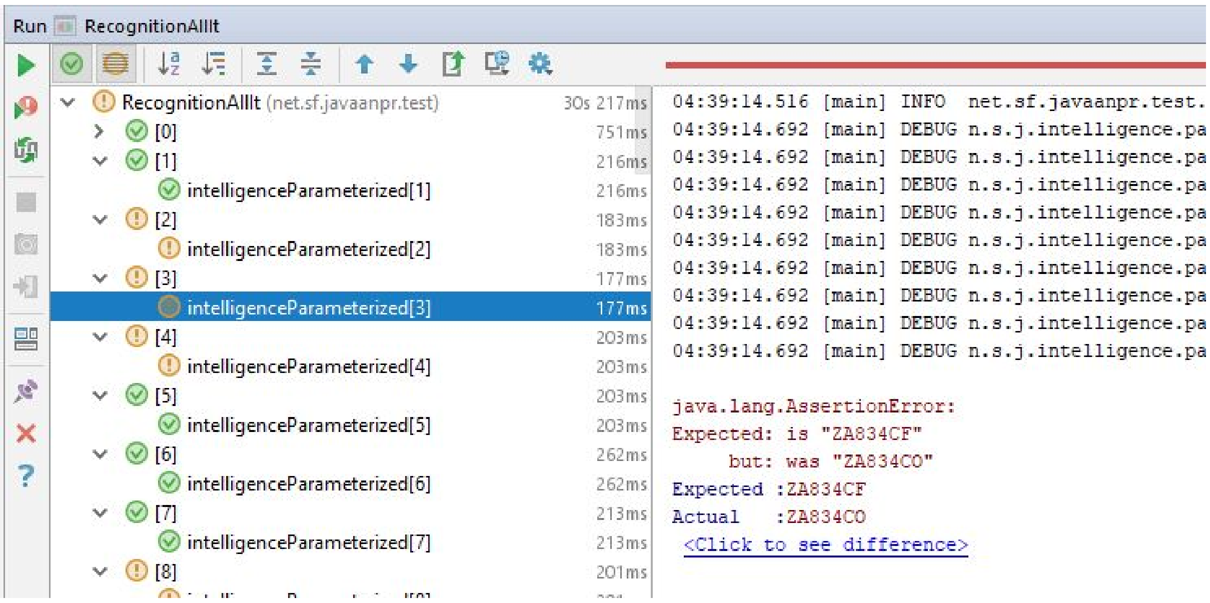
Marco Blum
Copenhagen business academy
03/04/2018
Software Development
Test.

https://github.com/Thug-Lyfe/test_midterm

Assignment 1)

● Explain the purpose of the Test (what the original test exposed, and what your test exposes)

The first test exposed that some of the images had errors in 44 out of 97, but gives cluttered console logs so finding out which had errors is a chore.
The parameterized test separates all of the images into their own test, showing all which gives errors separately. Making it alot more readable than previously.



```
RecognitionAllIt (net.sf.javaanpr.test)      30s 217ms
  [0]                                        751ms
  [1]                                        216ms
    intelligenceParameterized[1]             216ms
  [2]                                        183ms
    intelligenceParameterized[2]             183ms
  [3]                                        177ms
    intelligenceParameterized[3]             177ms
  [4]                                        203ms
    intelligenceParameterized[4]             203ms
  [5]                                        203ms
    intelligenceParameterized[5]             203ms
  [6]                                        262ms
    intelligenceParameterized[6]             262ms
  [7]                                        213ms
    intelligenceParameterized[7]             213ms
  [8]                                        201ms
```

```
04:39:14.516 [main] INFO  net.sf.javaanpr.test.
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa
04:39:14.692 [main] DEBUG n.s.j.intelligence.pa

java.lang.AssertionError:
Expected: is "ZA834CF"
     but: was "ZA834CO"
Expected :ZA834CF
Actual   :ZA834CO
 <Click to see difference>
```

● Explain about Parameterized Tests in JUnit and how you have used it in this exercise.

Parameterized tests are used to test data driven code. where a test can be run multiple times with many different inputs and outputs, none of which is hard coded and all of them are seperated.
I used Parameterized tests by simple making a new class with:

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

Where the parameters came from the results.properties file, which i made into a {key,value} ArrayList so it can be used in the parameterized tests. The test itself is ripped off from the ***intelligenceSingleTest()*** where it has been slightly changed to use the parameters instead of the hardcoded data.

● Explain the topic Data Driven Testing, and why it often makes a lot of sense to read test data from a file.

Data driven testing makes it so the tests are not hardcoded and that the data used can be changed and/or updated and/or extended, so the tests themselves don't go stale so to speak and the tests will be run multiple times with lots of different data.

● 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 first problem of the test is that it assumes that 53 tests are going to be correct. This makes the test almost irrelevant as it does not provide any new information to the tester, instead it only lets the tester know that nothing has changed or something has changed but not what. Instead all the images should be tested individually so that those that gives errors can be improved upon while those was correct can be tested to see if they change. The second part is using way too many console.logs for one test as each test has 10-20 lines and with 97 images that makes at least 1000 lines of text which is just a bad standard to have. making it all parameterized and data driven solves all these problems.

● The steps you took to include Hamcrest matchers in the project, and the difference they made for the test
In a maven project it is simple to add Hamcrest. All that is needed is to add Hamcrest as a dependency in the *pom.xml* file and rebuild. In this specific test i do not see any real improvement from using standard junit as it is a fairly simple test, but there are small changes in the error messages.

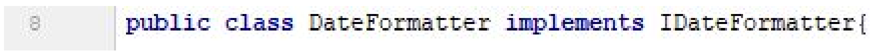
Assignment 2)

At the beginning the dateformatter used external API's for time zones, which is fine in practice, but for testing it is huge problem as you can't make sure it works 100% the same each time.

So first off we need to refactor according to the rules of dependency injection and inversion on control. Here we instead provide an instance as an input.
Therefore I chose to refactor the DateFormatter class so that it takes two inputs when initiasized. A String with the timezone and an instance of a java.util.date object, defining the current time.

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.

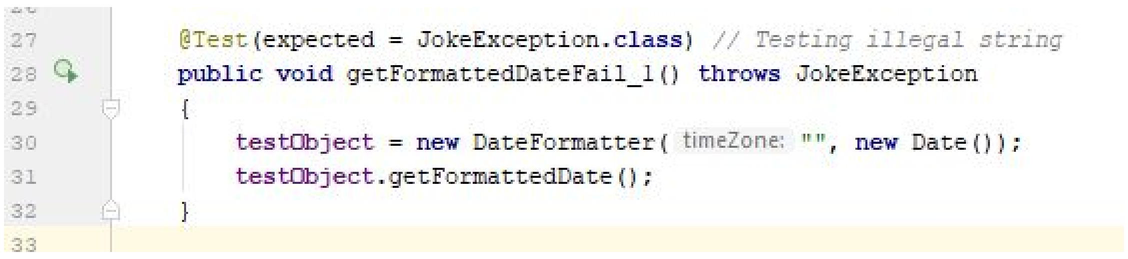
While this works great in practice, it is almost impossible to test since the state of the program is defined by outside influence. Therefore we need to refactor the method according to the according to the rules of dependency injection and inversion on control, where we instead provide some instance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



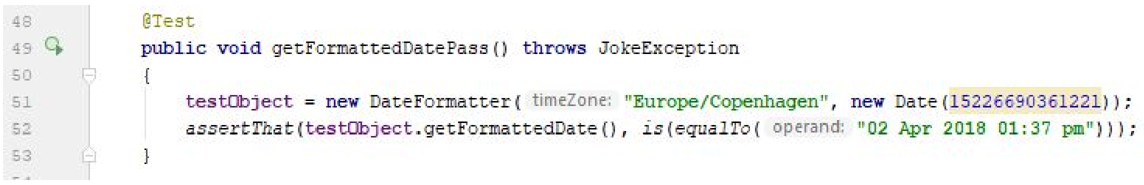
```
public class DateFormatter implements IDateFormatter{
```

The DateFormatter implements an interface making it easier to test.

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



```
@Test(expected = JokeException.class) // Testing illegal string
public void getFormattedDateFail_1() throws JokeException
{
    testObject = new DateFormatter( timeZone: "", new Date());
    testObject.getFormattedDate();
}
```



```
@Test
public void getFormattedDatePass() throws JokeException
{
    testObject = new DateFormatter( timeZone: "Europe/Copenhagen", new Date(15226690361221));
    assertThat(testObject.getFormattedDate(), is(equalTo( operand: "02 Apr 2018 01:37 pm")));
}
```

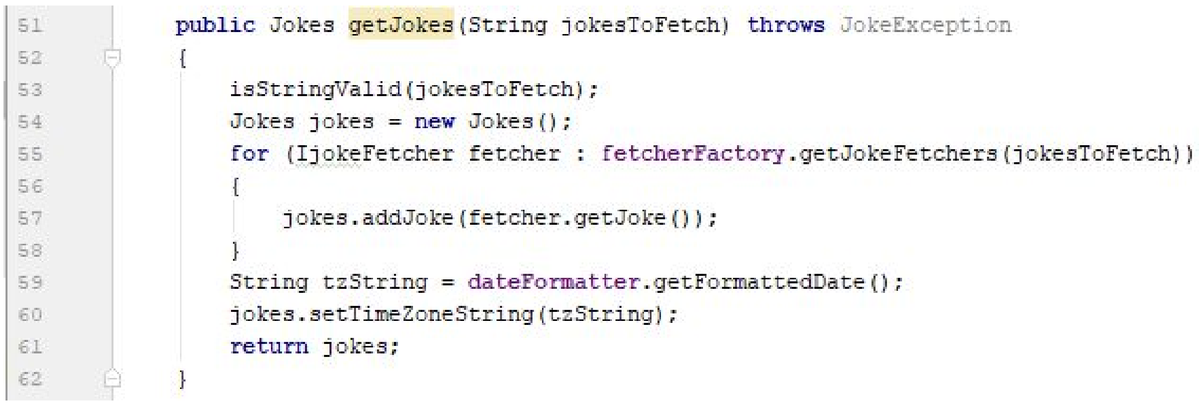
Two tests from the DateFormatterTest class.
Notice how the second test is designed so that the output is constant.

This also makes it so that i can hardcode a time for all my tests, so that i can ensure that it does not affect my output.

The *JokeFetcher* class had similar problems to the dateformatter class. First and foremost it contained four calls to external REST APIs, where we would like to limit our dependency during testing.

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, is to move all the logic associated with retrieving the jokes from the internet over to new classes, all of these gets implemented in the *IJokeFetcher* interface.

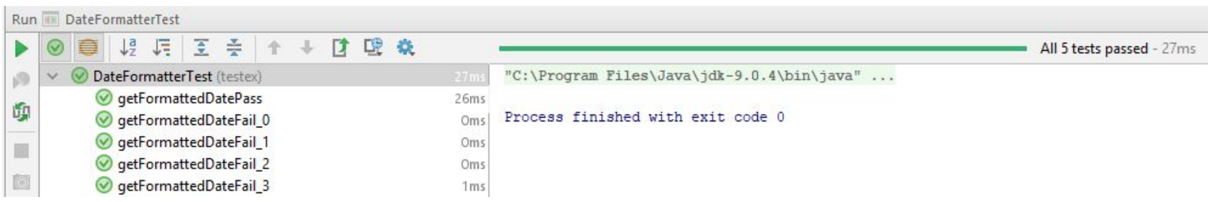


```
public Jokes getJokes(String jokesToFetch) throws JokeException
{
    isStringValid(jokesToFetch);
    Jokes jokes = new Jokes();
    for (IjokeFetcher fetcher : fetcherFactory.getJokeFetchers(jokesToFetch))
    {
        jokes.addJoke(fetcher.getJoke());
    }
    String tzString = dateFormatter.getFormattedDate();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

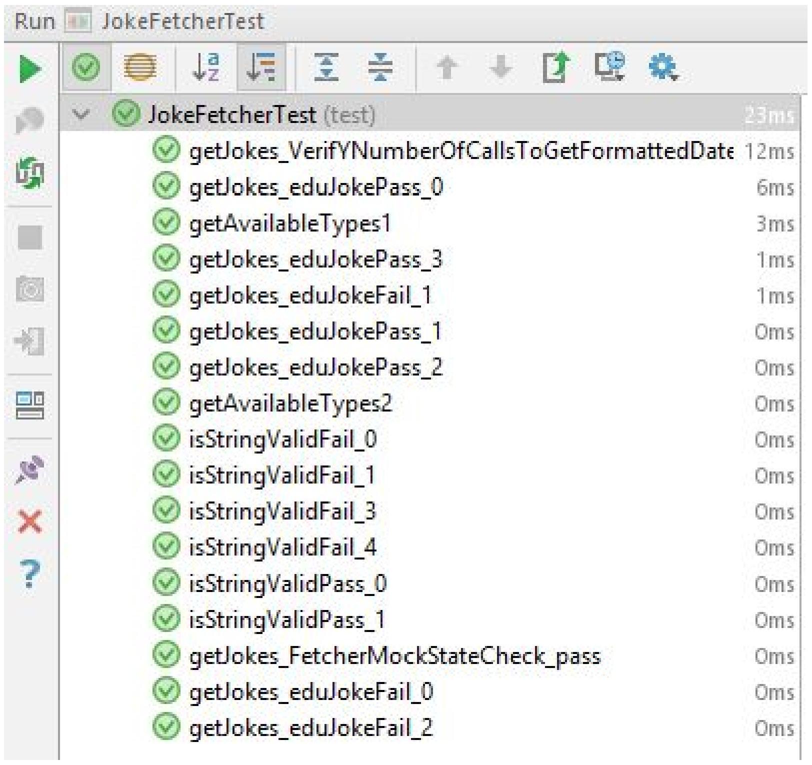
I decided to test both the DateFormatter and the *JokeFetcher.* The DateFormatterTest is pretty simple with 5 unit tests, 4 for illegal inpuit and 1 for verification that it works.

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                                          All 5 tests passed - 27ms
DateFormatterTest (testex)   27ms   "C:\Program Files\Java\jdk-9.0.4\bin\java" ...
  getFormattedDatePass       26ms
  getFormattedDateFail_0      0ms   Process finished with exit code 0
  getFormattedDateFail_1      0ms
  getFormattedDateFail_2      0ms
  getFormattedDateFail_3      1ms
```

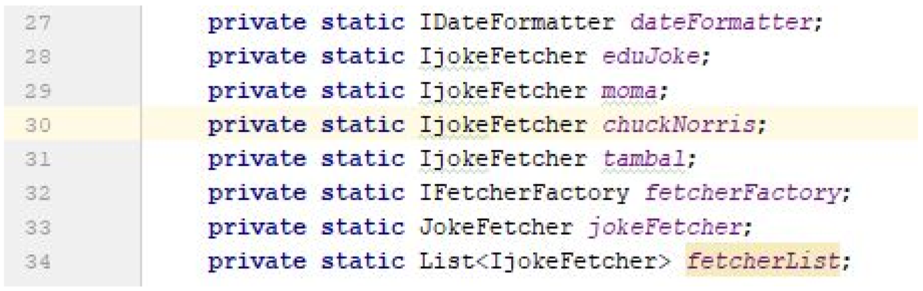


```
Run  JokeFetcherTest
JokeFetcherTest (test)                                23ms
  getJokes_VerifYNumberOfCallsToGetFormattedDate      12ms
  getJokes_eduJokePass_0                               6ms
  getAvailableTypes1                                   3ms
  getJokes_eduJokePass_3                               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
```

A screenshot of the DateFormatterTest class in action.

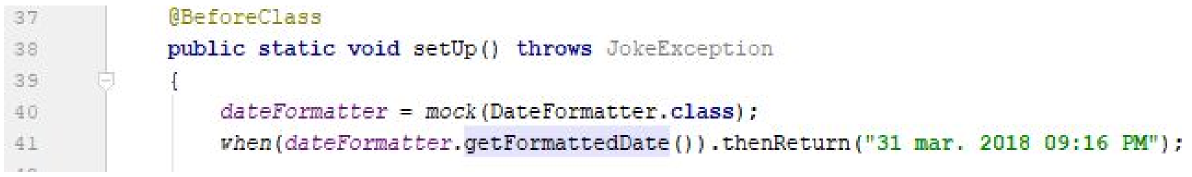
My Tests are all made with Junit 4, while Hamcrest is used for assertions. I did this because Hamcrest has slightly better error messages and improved functionality such as type checks and improved readability.

I used Mockito to create mock objects, which is essential for running the *JokeFetcherTest.*



```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;
    private static List<IjokeFetcher> fetcherList;
```

A view of all the variables that i mock

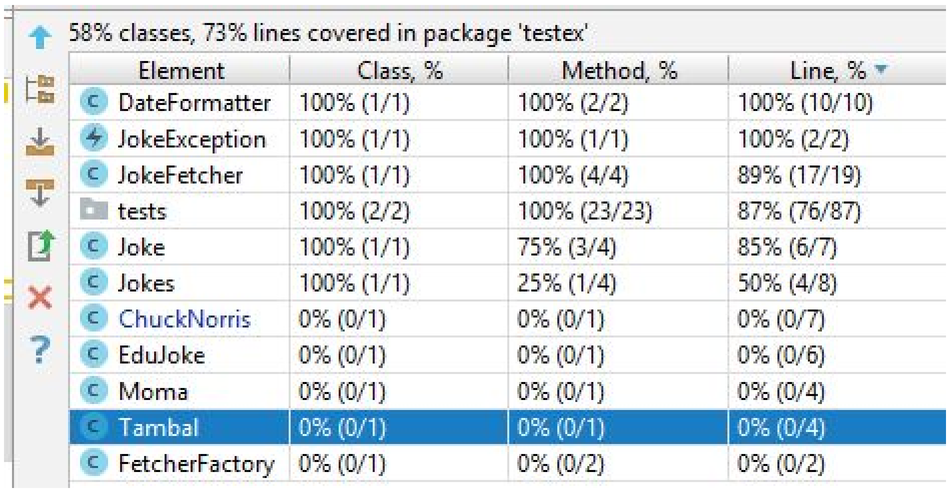


```
@BeforeClass
public static void setUp() throws JokeException
{
    dateFormatter = mock(DateFormatter.class);
    when(dateFormatter.getFormattedDate()).thenReturn("31 mar. 2018 09:16 PM");
```

The DateFormatter initialization with Mockito. the other variables from the picture above are initialized in the same way

Here I create the variables and then i initialize them as mocks and give them appropriate return values, for different operations.

I have not used Jacoco, as Intellij has a built in tool for code coverage.



58% classes, 73% lines covered in package 'testex'

Element	Class, %	Method, %	Line, %
DateFormatter	100% (1/1)	100% (2/2)	100% (10/10)
JokeException	100% (1/1)	100% (1/1)	100% (2/2)
JokeFetcher	100% (1/1)	100% (4/4)	89% (17/19)
tests	100% (2/2)	100% (23/23)	87% (76/87)
Joke	100% (1/1)	75% (3/4)	85% (6/7)
Jokes	100% (1/1)	25% (1/4)	50% (4/8)
ChuckNorris	0% (0/1)	0% (0/1)	0% (0/7)
EduJoke	0% (0/1)	0% (0/1)	0% (0/6)
Moma	0% (0/1)	0% (0/1)	0% (0/4)
Tambal	0% (0/1)	0% (0/1)	0% (0/4)
FetcherFactory	0% (0/1)	0% (0/2)	0% (0/2)

A screenshot of the test coverage where it can be seen that most of my effort has been used on the DateFormatter and the JokeFetcher classes