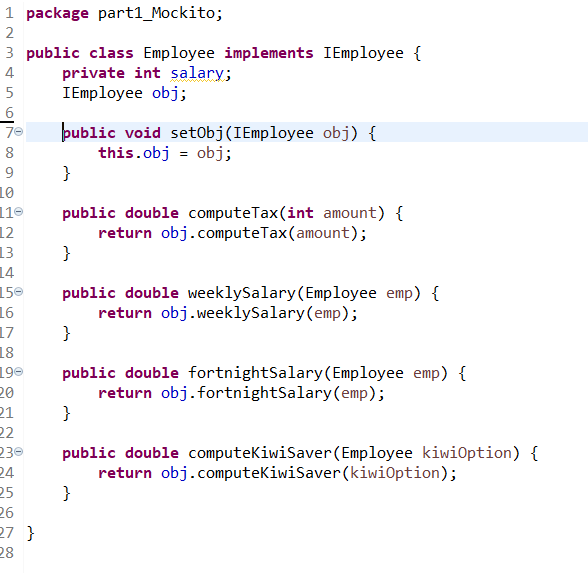
|  |
| --- |
| Wellington Institute of Technology |
| IT7320 – Development and Testing of Software |
| Assignment 3 – Mockito, Jenkins |

|  |
| --- |
| Dan Mota ID: 2150708  9-26-2018 |

# Part 1 – JUnit & Mockito

The exercise starts with the creation of the class Employee and interface IEmployee, as well as the appropriate implementation of the interface methods using the keyword ‘implements’ on the Class Employee definition.

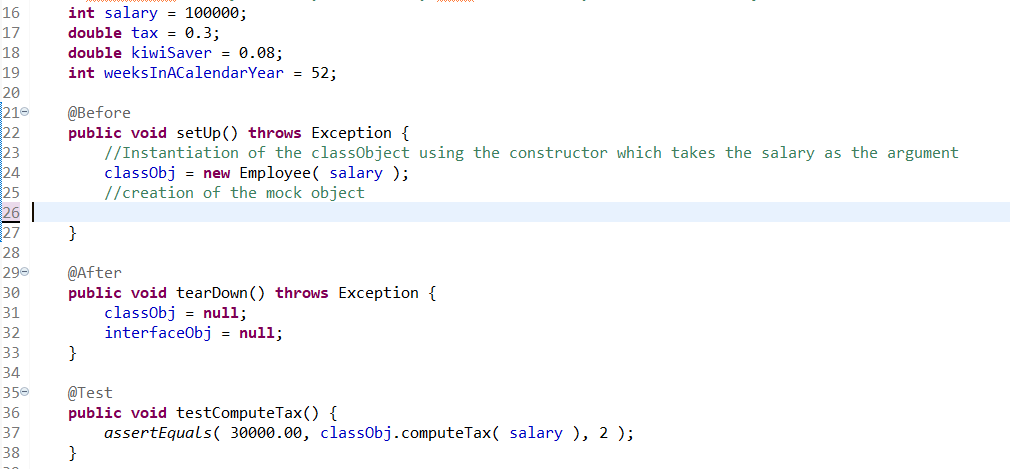
Unimplemented methods were added, an object of IEmployee interface was created which is used to call its own methods as thus:



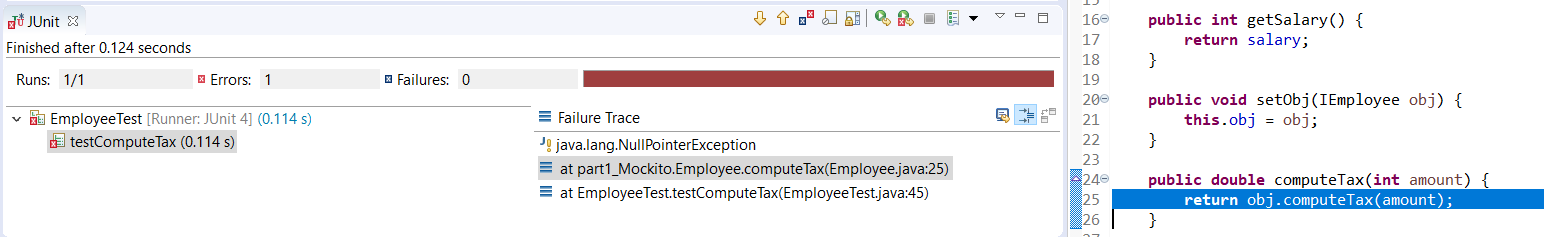
A new JUnit test case for class Employee was then created. Objects of class Employee and interface IEmployee were created, and assumed test variables were initialised: salary of type int as 100,000, tax of type double as 0.3, kiwiSaver of type double as 0.8, weeksInACalendarYear of type int as 52.

In the setUp method the class object is instantiated using variable salary as the parameter for the constructor.

A test is created where the expected value of compute tax is 30,000 based on our assumptions: salary = 100,000 and tax = 30%.



Running this test results in a NullPointerException, given that the method called executes the method inside the interface IEmployee which is a skeleton code.



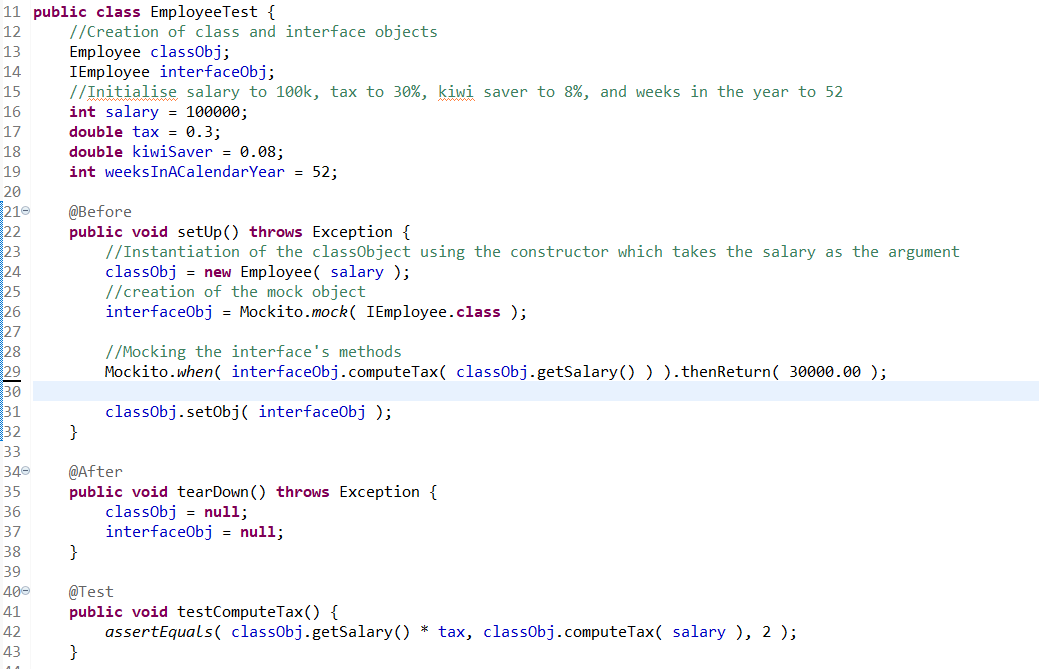
## Why Mock?

“The purpose of mocking types is to sever dependencies in order to isolate the test to a specific unit.” [1]

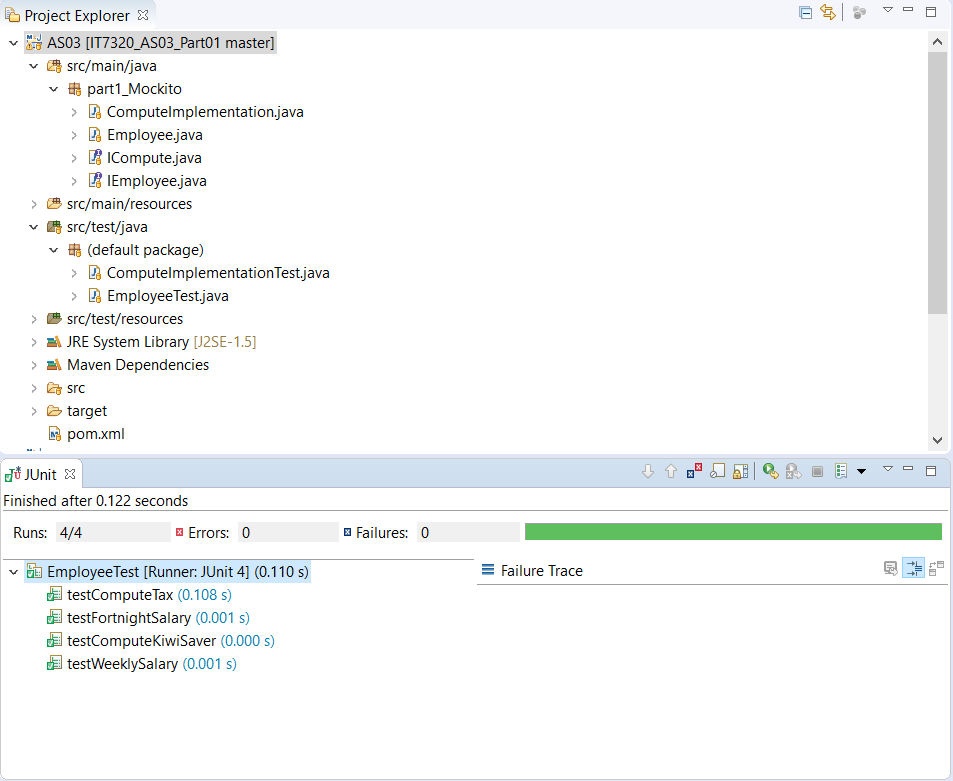
An interface was selected because it gives enough a degree of abstraction that is necessary when mocking. Classes are concrete and can be tested. What an interface does and how it works is often considered beyond the knowledge of a developer.

Likewise, parts of the system cannot be disconnected for the purposes of error fixing/patch creation in real life. Code that performs changes through a network – i.e. database and server connections – must be mocked to avoid any possibility of corrupting data or interfering with business processes.

Lastly, these connections are unit dependencies – and as stated in the first line of this section – dependencies must be severed in order to properly conduct a unit test.



As can be seen in the code above, a mock object is created in line 26. In line 29, the processed data return for the method computeTax is mocked. Given the salary is 100k and the tax rate 30%, the expected result is 30k as a double.



The same process of mocking method calls is carried out for the remaining methods of IEmployee interface: fortnightSalary, computeKiwiSaver, weeklySalary. Test cases were also created for each of the methods by making use of Mockito’s when().thenReturn() static method returning the appropriate expected return for each interface method.

3,846.15 is returned for fortnightSalary ( 100,000 / 26 )

1,538.46 is returned for computeKiwiSaver ( 100,000 / 52 \* 0.8 )

1,923.08 is returned for weeklySalary ( 100,000 / 52 )

The image above illustrates the test result for all our test cases. Methods successfully mocked, and assertions successfully performed.

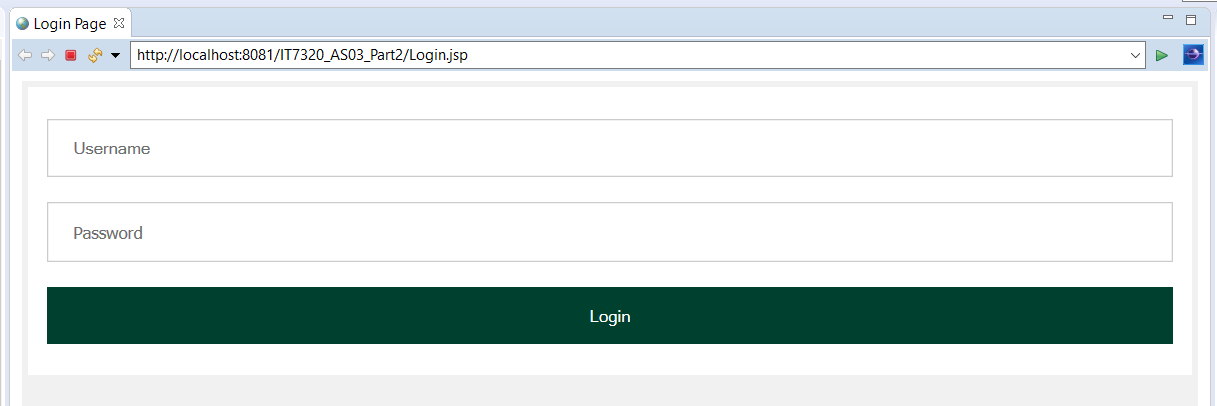
# Part 2 & 3 – Jenkins & TDD

For part 2&3 a web application is created. The username and password required for the application needs some rules. The first standard rule for passwords is the minimum length, as often seen in website and software registration processes, a minimum of 8 characters is necessary. Next, we need to define the rules regarding the legal structure of a username and password. Simply setting this rule as alphanumeric would be too ~~noobish~~ simplistic regarding security. Therefore, the IBM structure was selected:

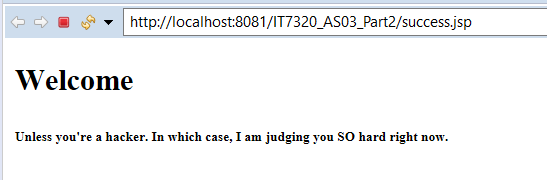
|  |  |
| --- | --- |
| ***Input Type*** | ***Valid Characters*** |
| Passwords and IDs\*  \*(apart from the characters described, IDs may also possess Commercial ats { @ }) | Lowercase characters { a-z }  Uppercase characters { A-Z }  Numbers { 0-9 }  Exclamation point { ! }  Open parenthesis { ( }  Close parenthesis { ) }  Dash { - } (*this character is not supported as the first character in the user ID or password*)  Period { . } (*this character is not supported as the first character in the user ID or password*)  Question mark { ? }  Open bracket { [ }  Close bracket { ] }  Underscore { \_ }  Grave accent { ` }  Tilde { ~ }  Number sign { # }  Dollar sign { $ }  Circumflex accent { ^ }  Ampersand { & }  Asterisk { \* }  Plus sign { + }  Equals sign { = } ~~HOW FUN!~~ |

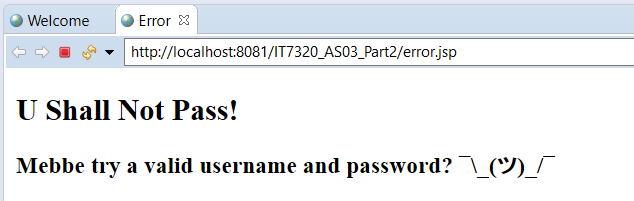
Also according to IBM[2], user IDs cannot be longer than 200 characters long, and passwords are limited to 128 characters. For the purposes of this exercise the minimum length of a user ID will also be set to 8.

Now that the rules are set, the general configuration of the Maven project, Jenkins and the simple HTML code for our Login Page is created:



Followed by the Success and Error pages which our webapp should take the user to based on username and password inputs:

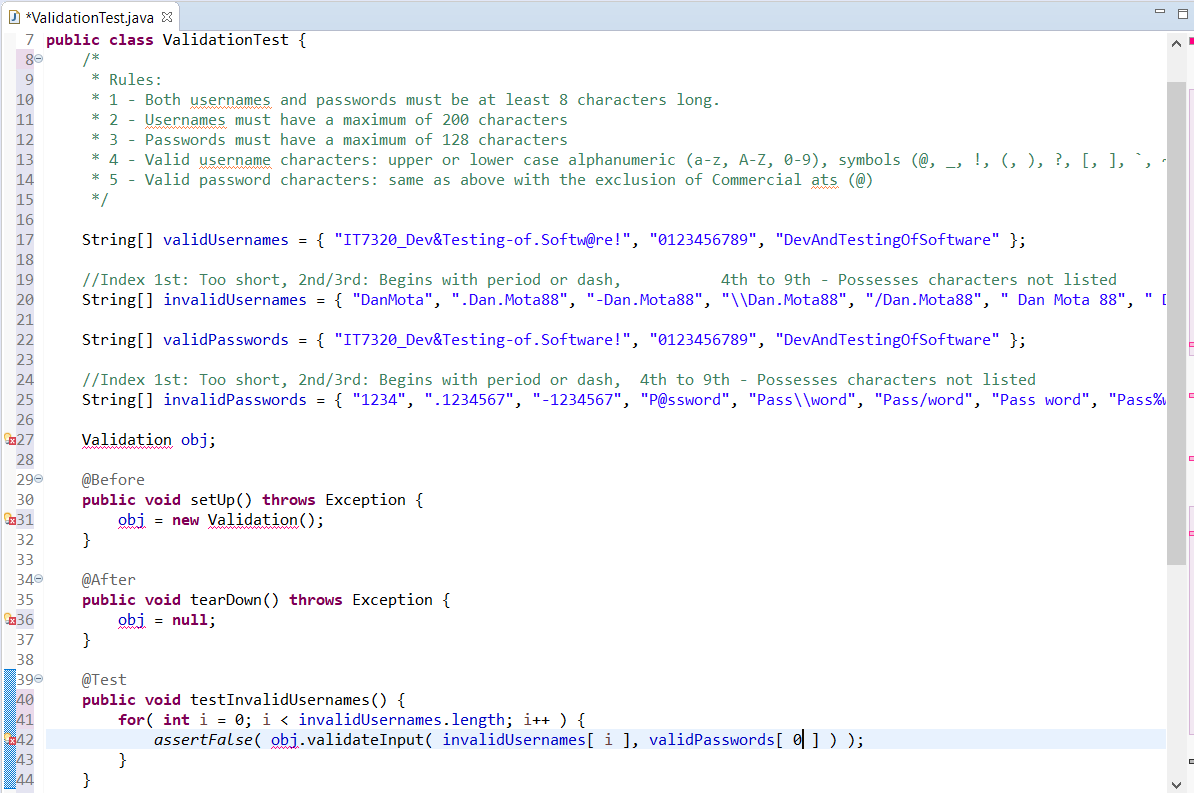




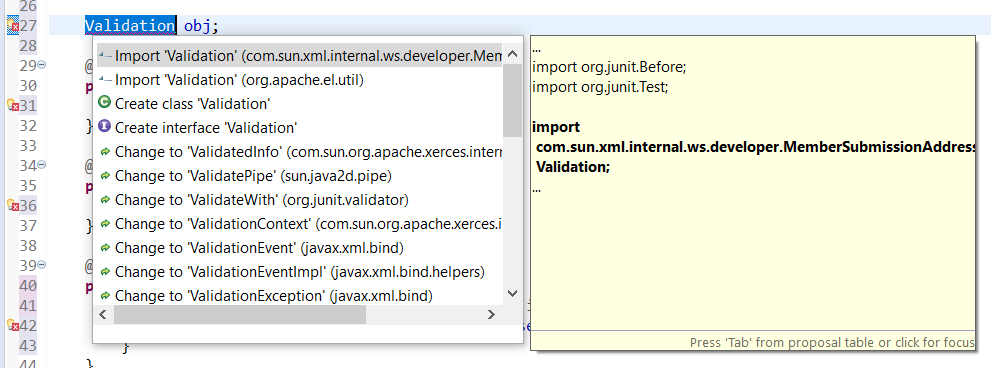
With those steps concluded, a servlet must be implemented to handle the Login page submits. During this step of the project, the doPost method is modified to send the user to the Error page, then modified once more to redirect to the Success page as a manual test to ensure that Tomcat, the JSPs, and the Servlet are successfully integrated. ~~Take my word for it?~~

## Test Driven Development

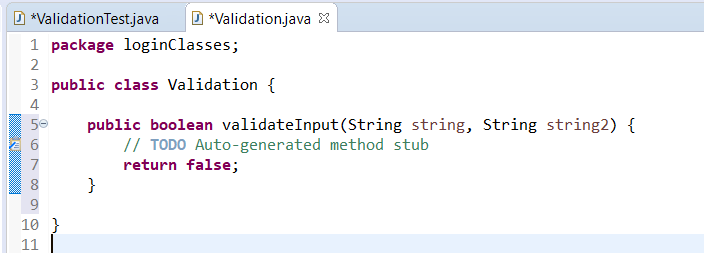
The initial class Validation that was created for part 2 is deleted to be rewritten following a Test-Driven approach. Next, a JUnit test case called ValidationTest is added for the project. Test data following the rules defined at the beginning of this section is assigned to string arrays, as well as the first test method:



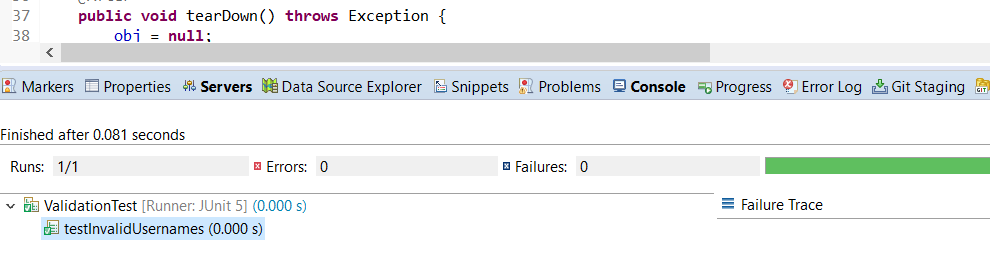
As seen above, Eclipse flags an error which cannot be resolved as the desired class does not exist:



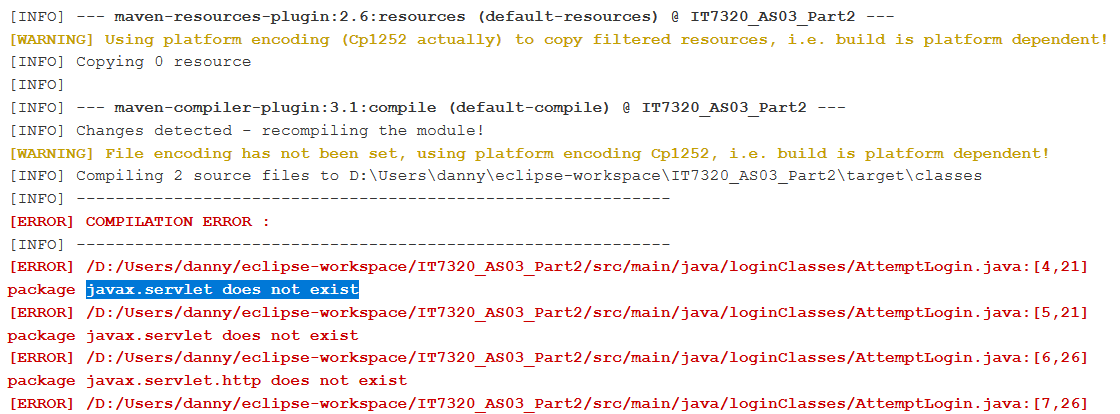
A blank class and skeleton code are auto generated to run our test.



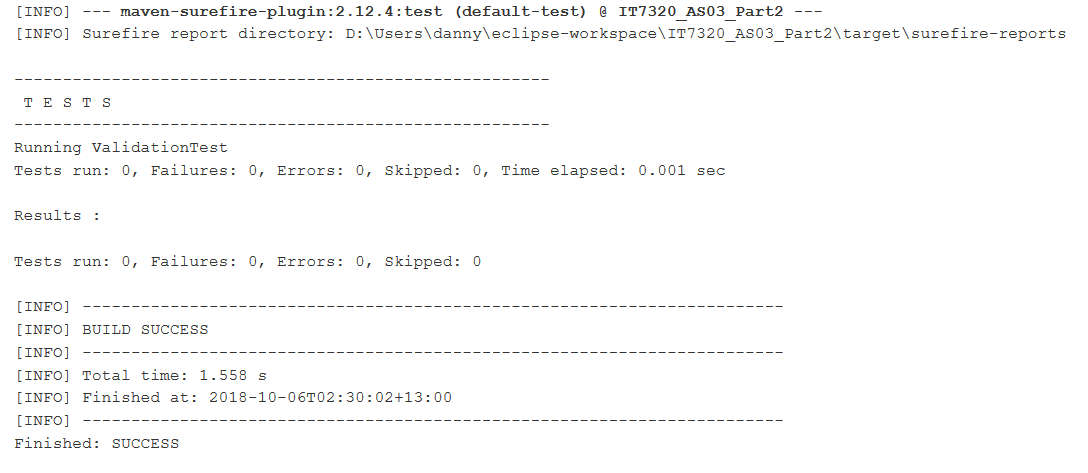
Logically, if we run the test as is it will pass as the method returns false.



At this stage we visit Jenkins and perform a build. That’s when things get interesting. As seen above, our test method is called testInvalidUsernames. When performing a build, Jenkins fails:

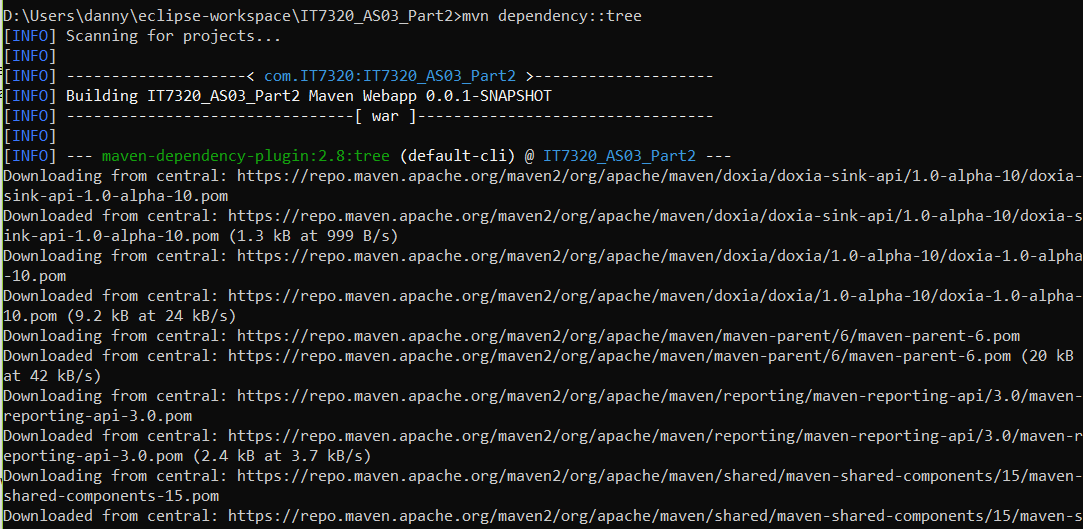


Interestingly enough, renaming the test method back to the default test(), saving, restarting the server, and reattempting the build results in a success. However:



No tests are run. ~~BUTLER! YOU’RE FIRED!~~ At this point a commit to GitHub is performed. Likewise, running the ‘mvn test’ and ‘mvn verify’ commands resulted in a "Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test" error.

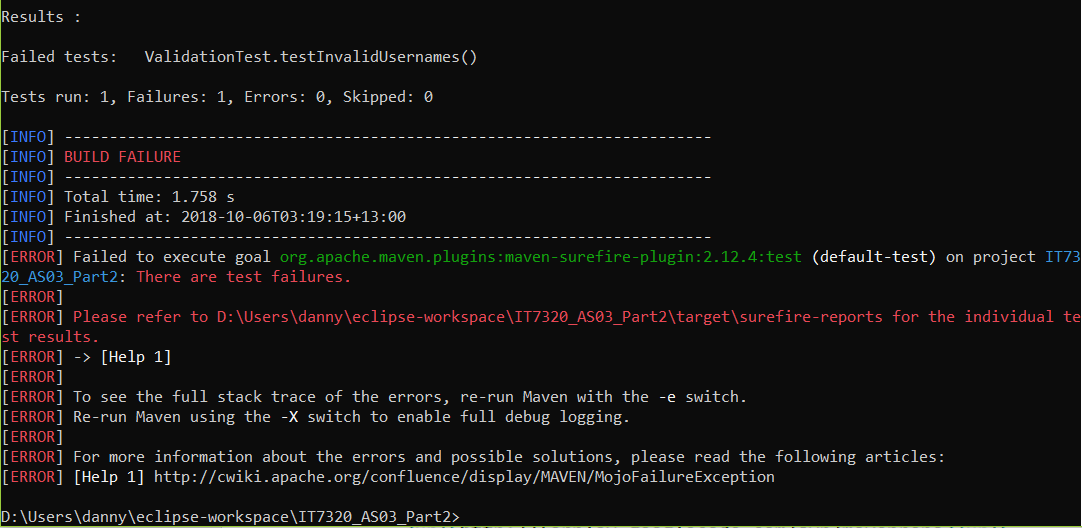
After a quick online research some suggestions to solve the build problem such as running the command ‘mvn dependency::tree’



As well as the addition of the following repository to the project’s pom.xml:



Re-running the ‘mvn test’ command resulted in the same failure:



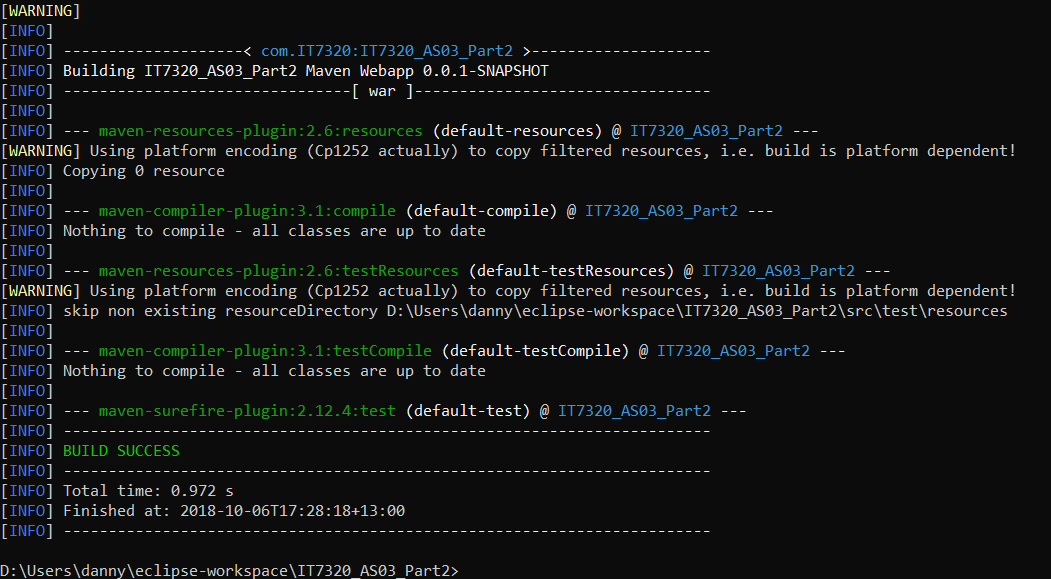
As well as right clicking the project > Maven > Update resulting in the same Build failure above.

The problem was solved by taking the following steps:

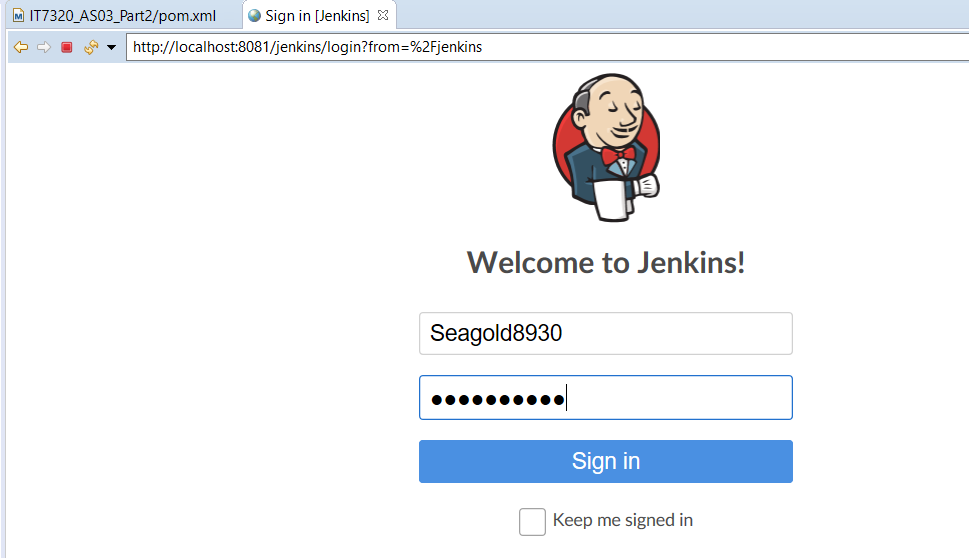
* Moved the test class into a new package called tests under the ‘src/main/java’ directory in Eclipse (this alone won’t fix the surefire issue)
* Next, the pom.xml file was opened and all of the ~~crap~~ extra dependencies were deleted, leaving only the repository shown above, the JUnit dependency – edited to version 4.12 – and inside the build > plugins tag the surefile plugin was added and configured to use JUnit 4.12 as thus:



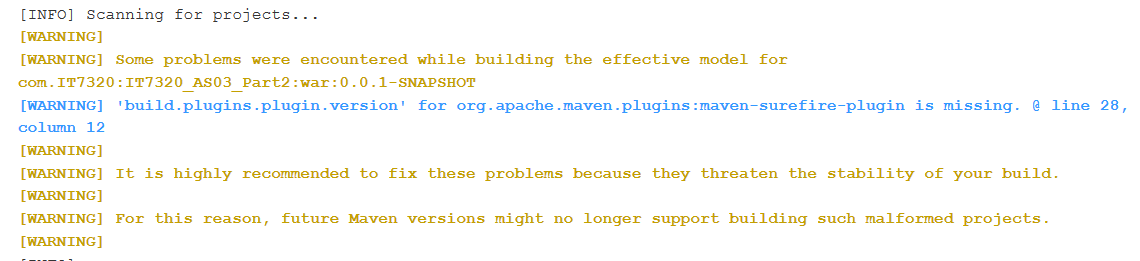
* Changes were saved, the project directory opened, and the command prompt through the window.
* The command ‘mvn test’ executed



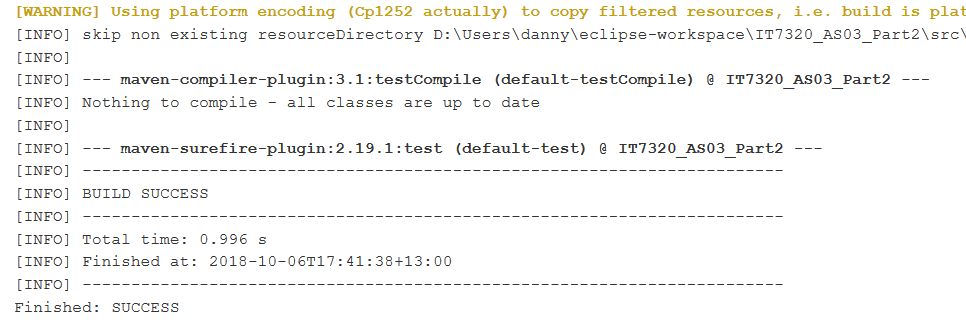
Lastly, we test the new configuration on our Tomcat/Jenkins



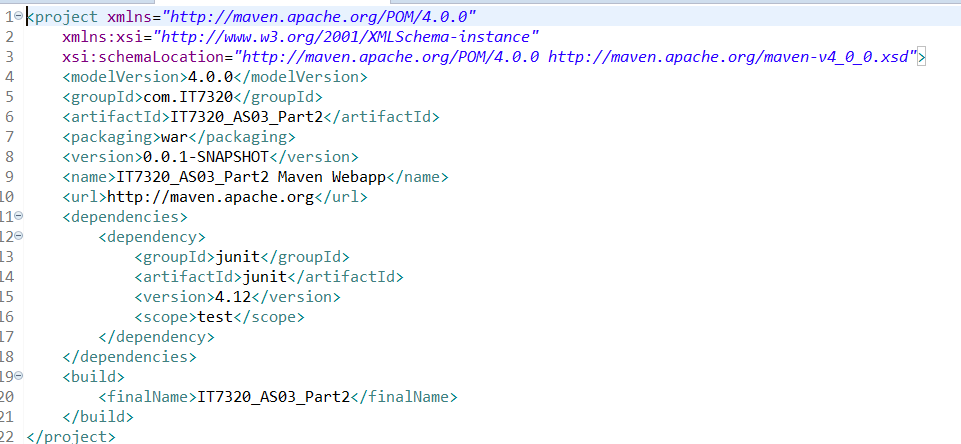
A new build for the project is started, and completed successfully with one important warning:



Although the build errors were resolved, a new problem can be seen. In the Maven build in the command prompt, no tests were run. The same behaviour can be observed in Jenkins:

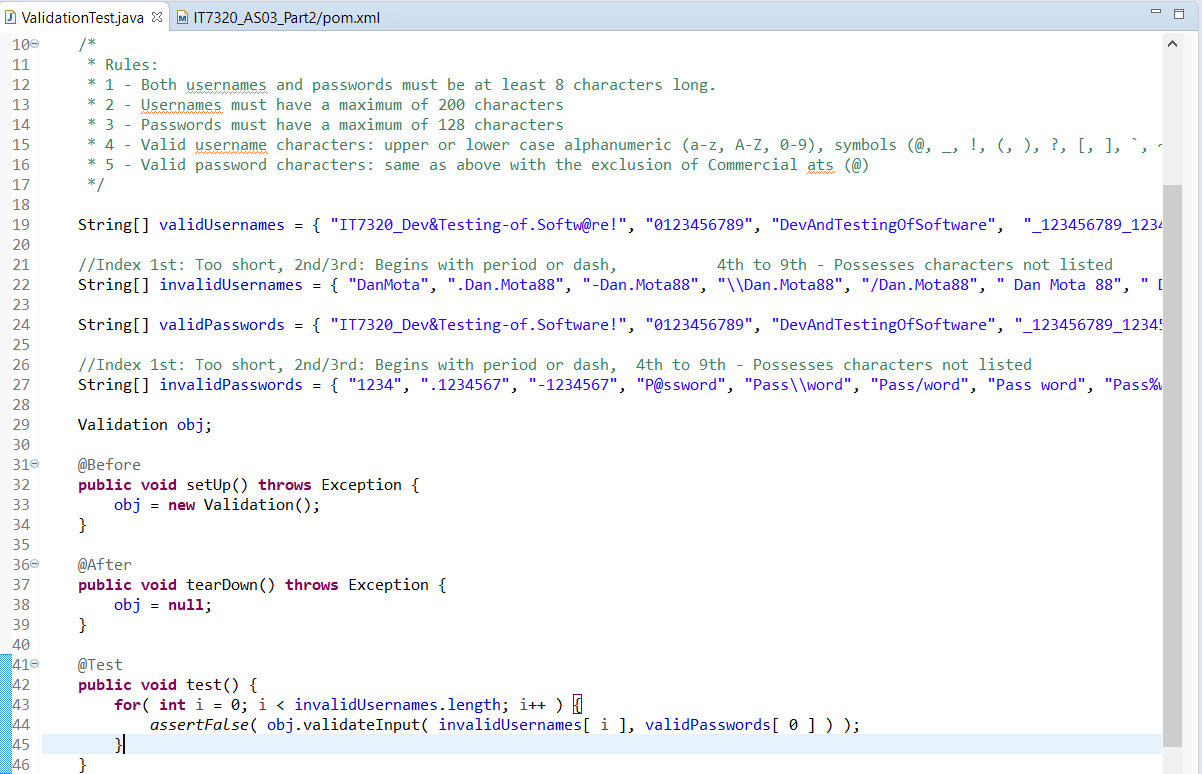


At this stage, it becomes apparent that the problem has become too complex, and our approach to resolve it not effective. Therefore, all addons in the pom.xml are deleted leaving only the JUnit dependency version 4.12. As well as our test case, contents were backed-up and the ValidationTest file deleted.

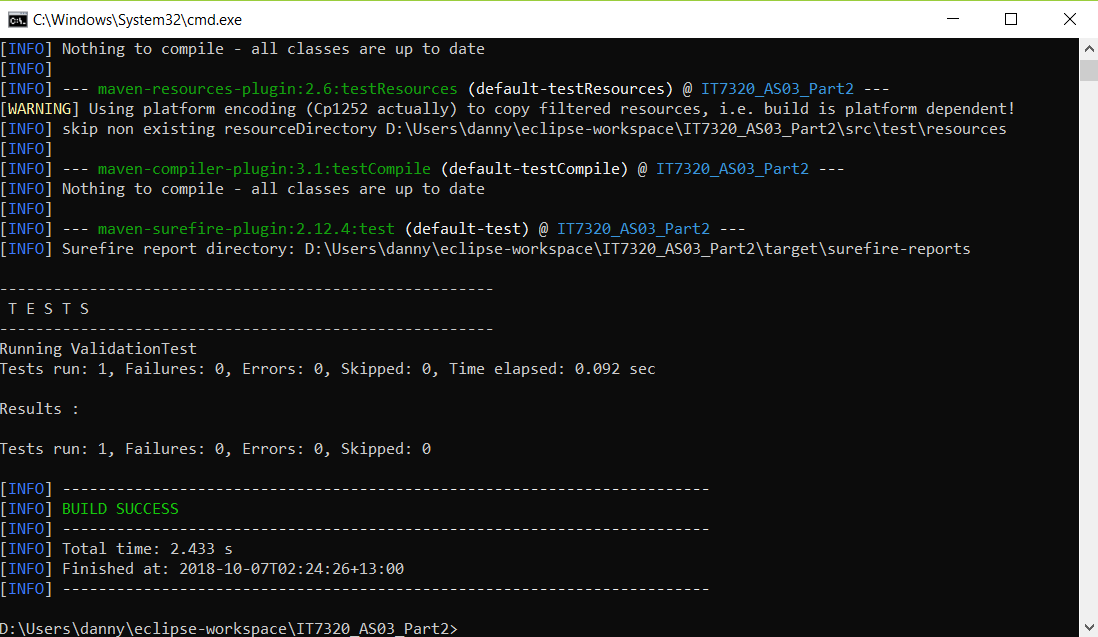


#### Resolving Maven and Tomcat Jenkins Problem

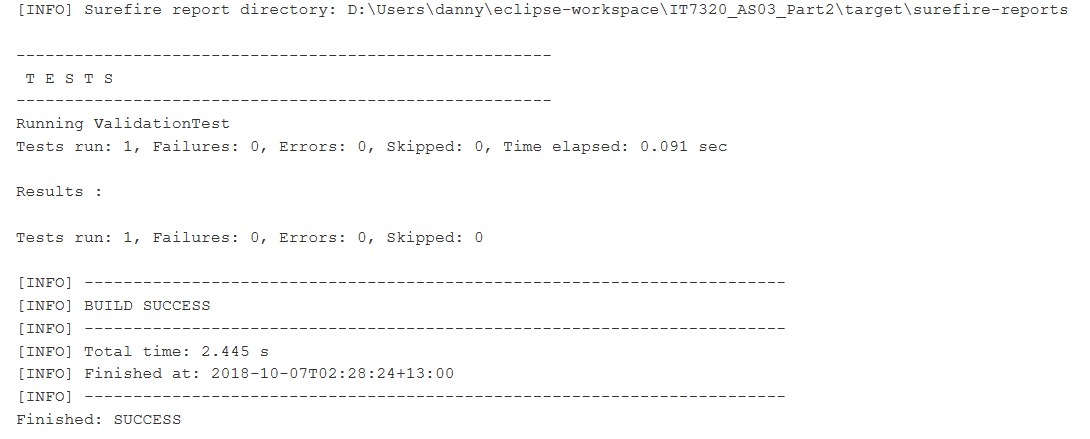
After deleting our test case, a new *Class* is created under ‘src/test/java’. A new test method created from scratch, including the setUp and tearDown methods. At this point the JUnit imports are performed in Eclipse and the backed-up contents of our deleted test case copied into the new file.



The test is run on Eclipse to ensure it works. And finally, we reopen the command prompt in our project directory typing the ‘mvn test’ command once more.



Fully functional this time. And also, in Jenkins:



A new commit and push to Git is then performed.

# Appendix A – GitHub

## Profile - Dan

The author’s profile and GitHub project for the part 1 exercise can be found [here](https://github.com/Seagold8930/IT7320_AS03_Part01).

## Source Code

The source code for the part 1 exercise can be accessed on GitHub [here](https://github.com/Seagold8930/IT7320_AS03_Part01/tree/master/AS03/src).

# Appendix B – Resources

[1] <https://stackoverflow.com/questions/2665812/what-is-mocking> retrieved on 19/09/18 at 3:51PM

[2]<https://www.ibm.com/support/knowledgecenter/SSV2LR/com.ibm.wbpm.imuc.doc/topics/rsec_characters.html> retrieved on 02/10/18