**Top Right Corner**

Testing methods and approaches

We started by writing automated unit tests for every class except for the PopUpWindow class, any classes that simply consisted of a single enumerated type, the main class (it would very hard to automatically test the main class as it would have been equivalent to automated system testing) and the Desktop Launcher class. Unit tests were divided into a test case for each class that contained unit tests for every public method in the class under test. We tried our best to follow a test driven development process and write these unit tests before even starting to write code. Unfortunately in some places we fell short of this mark, mainly when it came to testing the GUI-oriented classes that depended on LibGDX as at the time we did not know how to write the tests for these classes. Ultimately we also wrote tests that simulated the use of the GUI elements that e.g simulated the clicking of buttons and checked the contents of labels.

We used Junit to write all automated tests and JMockit to mock out any dependencies when unit testing (create fake versions of objects that the class under test depended on that would always behave in a specified way[1]). Without mocking our unit tests would not have been true unit tests. A unit test is meant to test only one thing[2], if we did not mock out these dependencies we would also have been testing the functionality of the other objects[1]. Using JMockit also allowed us to be sure that mocked objects were being used correctly by the class under test ensuring that the correct methods from the mocked object were called with the correct arguments, thus increasing the power of our tests. In many cases we wrote several tests to test one method using the same input data to test different aspects of the method call such as the result produced and the methods of various mocked objects that were called. Although it massively increased the number of tests, should any one test fail it will be far easier to find out exactly what went wrong, making it far easier to debug our code. It's worth noting that in some cases with the GUI-oriented classes our unit tests were not true unit tests as the classes had several dependencies that we could not mock as the objects were created in the classes themselves and could not be passed via a constructor or setter method. For more information relating to this see the GUI test cases.

Later we wrote integration tests, testing many objects together to ensure that they worked well together, we did this as some behaviour only becomes obvious once several components have been combined together[3]. In many cases we took unit tests that involved several mocked objects and rewrote them using 'real' (not mocked) objects, saving time and ensuring that the same functionality was tested using real objects.

Finally, we performed some manual system testing, testing the system 'as a whole' rather than individual components[3]. We did this to be sure that all the components behaved properly together (some behaviour only becomes apparent once components have been combined[3]). Automated system testing is very difficult[3] and the tests would have been very complex due to the large sequence of operations that would have to be performed to simulate e.g. a single player's turn, we would likely made mistakes when writing them. We also wanted to convince ourselves that the GUI worked correctly as the automated GUI tests didn't seem as robust as the automated 'back end' tests and some elements (such as the location of a window) could not be testing due to apparent limitations with the headless LibGDX backend that we were using. The large quantity of automated testing that we'd carried out meant that we didn't have to carry out a large number of manual tests as e.g. the majority of the validation tests etc had already been fully tested.

Automating most of our testing made regression testing (retesting the code when changes were made[4]) far easier as the tests can be run in seconds. This allowed us to continually refactor our code whenever we saw something that could be improved (as we stated we would in the methods document of assessment 1) without fear of breaking anything or spending time manually testing code. This made us more comfortable refactoring our code and likely improved its overall quality[3]. The team that takes on our project will also benefit from this as they can easily retest after they've made changes. Its also worth noting that the automated tests act as a form of documentation, if other teams don't understand something we've done they can read the tests to help them understand what the code is supposed to do[3].

Actual tests run and summary of the results

A total of 962 automated unit and integration tests were run and all were successful. For every method in every backend class that was automatically tested we divided the inputs into sets for which the code should behave similarly. We then made sure (as recommended by Sommerville[3]) to test at least 2 values from the middle of every group and all values that lay right on the edge of each set (extreme values e.g. the maximum and minimum values that a given method can accept) as well as all values that would cause exceptions to be thrown. We did this as generally programs will respond similarly for all data in a set[3] and we only had a limited amount of time in which to perform our testing and therefore needed to ensure that we didn't spend too long testing anyone one method. For every gui-oriented class we did the same except instead of calling methods with various inputs, we simulated the use of the various on screen controls using them to provide and submit various inputs for testing. Our automated tests involved a mixture of both black-box (considering only the interface to our code and not its internals[5]) and white-box testing (also considering the code itself[5]). We tried to focus largely on black box testing as we felt that when white box testing we may make too many assumptions about the way in which our classes operate and such tests could fail if the implementation changed. However sometimes white box testing was useful as it allowed us to check that the correct methods from mocked objects were called (see the previous section), allowing us to be sure that our code was performing the correct operations even if it relied on the code in other objects.

We also carried out 89 manual system tests and all of these were also successful. We didn't focus to heavily on the system testing as the vast majority of our code had already been tested by this point (as stated in the previous section). These tests focussed on using the system as a user would use it and were all black box tests. The tests aimed to ensure that the correct actions were performed when users performed valid actions e.g. clicking an unacquired plot during the plot acquisition stage would cause that plot to be acquired by them and either nothing happens or an error message was produced when a user performed an incorrect action e.g. nothing happens when you click one of the AI player's plots.

Unfortunately we were unable to unit test the PlaceRoboticonMenu class, we are unsure why (perhaps due to limitations with the headless LibGDX backend that we were using to run our tests) but every time we attempted to instantiate an object of this class in a test case an error would be thrown that we couldn't decipher. However, this class was very simple and its instantiation (opening of a menu) was tested in the PlotManager test case and functionality was fully tested during manual testing. Ignoring this exception we feel that our tests were fairly complete. As we started from the simplest level with unit testing and tested every public method (and/or button press for gui classes) using normal, extreme and erroneous data we can be sure that every line of code has been executed at least once under both normal and extreme conditions and every exception that could be thrown by our system has been thrown. By then going up a level and performing integration tests we can be sure that the objects in our system interact well together when combined to form composite components and that there is no unexpected behaviour that results from combining these objects. Finally by testing the system as a whole via system testing we can be sure that all these composite system components interact well together and that once again is no unexpected behaviour revealed by combining said components. By testing in this way working from the level of single methods up to an entire system we were able to test our system far more completely than if we had just performed system testing as we can be sure all the code has been thoroughly tested rather than just that which is easily testable using the user interface. For example players cannot try to buy more ore than the market has in stock (as the spinbox used to select the quantity of ore that they wish to buy does not allow them to). If we had not also performed unit testing we would not have been able to test what happens if a user does manage to try buy more ore than is available, if the ore buying code was reused later in another project or the system modified in a way that did allow users to try and buy more ore than is available we could not be confident that our code would work as expected. It is worth noting that most of our non-functional (other than requirement 2.1), user interface and constraint requirements were untested as we could not develop precise tests to test them. We do however feel that we came up with a good argument that shows that requirement 4.1 holds in the GUI report and we are certain that requirement 3.1 holds as we have played the game many times on the computers in CSE 069/070.

We did not unit test the DesktopLauncher class as its extremely simple and its sole purpose is to launch the game so it would be very obvious if it was non-functional.

A JavaDocs that explains what all of our automated tests do and their expected outcomes can be found here:
https://sepr-topright.github.io/SEPR/documentation/assessment2/testJavaDoc/index.html

Our manual test plan and results can be found here:
https://sepr-topright.github.io/SEPR/documentation/assessment2/ManualTest.pdf

Our automated test report can be found here:
https://sepr-topright.github.io/SEPR/documentation/assessment2/test/index.html

Our traceability matrix showing which tests test which requirements can be found here:
https://sepr-topright.github.io/SEPR/documentation/assessment2/matrix/matrix.html

Link for Junit: http://junit.org/junit4/

Link for JMockit: http://jmockit.org/

# References


[1]P. Nijssen, "Mock your Test Dependencies with Mockery", *SitePoint*, 2014. [Online]. Available: https://www.sitepoint.com/mock-test-dependencies-mockery/. [Accessed: 22- Jan- 2017].

[2]"Properties of Good Tests: A-TRIP – cavdar.net", *Cavdar.net*, 2008. [Online]. Available: http://www.cavdar.net/2008/10/25/properties-of-good-tests-a-trip/. [Accessed: 22- Jan- 2017].

[3]I. Sommerville, *Software engineering*, 10th ed. Harlow: Pearson, 2016, pp. 226-245.

[4]"What is Regression testing in software?", *Istqbexamcertification.com*. [Online]. Available: http://istqbexamcertification.com/what-is-regression-testing-in-software/. [Accessed: 22- Jan- 2017].

[5]"Differences Between Black Box Testing and White Box Testing – Software Testing Fundamentals", *Softwaretestingfundamentals.com*. [Online]. Available: http://softwaretestingfundamentals.com/differences-between-black-box-testing-and-white-box-testing/. [Accessed: 23- Jan- 2017].