Software Testing Report

Testing Methods and Approaches

In order to ensure our solution is working effectively to the level dictated by our requirements, we will be using a mixture of white box and black box testing as determined by the functionality being tested. Code that may be unit tested must have the correct unit tests written for it, whilst also making sure that these unit tests are passed prior to the deadline of Assessment 2. This will include both the conflict resolution function, and the unit allocation function. We feel this is the most time efficient method available to us, as alternatives such as walkthroughs, or manually running the program through will take longer when testing the solution with different values. Unit testing also ensures that the code that is written actually runs correctly, as static testing methods may fail to reveal unpredicted errors at runtime that may not be obvious or accounted for. We will be using JUnit [1] to unit test our solution, as it is a well documented unit testing platform that many of our group are familiar with using.

However, due to the nature of some areas of the solution, it is not possible for everything to be unit tested. An example of this is the map, as unit tests are unable to simulate the inputs effectively. In this case, we will instead be making use of a black box approach to testing, focussing on how the solution meets the requirements it aims to address. Sections of code that cannot be unit tested will be walked through by a member of the team, simulating the behaviour of a user and thereby operating the program through the same channels that they would (for example mouse or keyboard control). Other than the change in the way the test is executed, the procedure for testing in these circumstances will largely remain the same, with test cases being used to ensure correct test protocol is followed and relate the test carried out back to the requirements.

One important thing to note is that in some cases it may not be possible to use a pure black box approach due to the lack of GUI. Given these circumstances, a grey box approach must be taken, with certain areas being simulated based on knowledge of the code to allow an approach that mimics black box to move forward. This is likely to include areas such as conflict resolution, as due to the limited scope of the program we are currently able to implement, it will not be possible to test this through the GUI.

Due to the areas of the solution we are required to implement, our GUI will not be complete enough for it to be relevant for us to begin testing non-functionally for this assessment. Though we aim to address non-functional tests in the future, it is largely an area that will remain untested at this point. One exception to this rule is checking that we have met our non-functional requirements with the current iteration of the solution.

As a result of the sections of the solution we are required to implement in this stage of development, we feel that integration testing will largely not be necessary. This is because the majority of the areas of the solution we have implemented require further implementations from later development stages to allow this to take place.

To establish to what extent our solution meets the requirements we elicited, we will be making use of test cases and a traceability matrix. This gives us a clear indication of which requirements are proven to be tested for, and ensures that no test cases are created without a specific requirement in mind, thereby reducing redundancy whilst being as thorough in our testing as possible. Test cases will form the structure of our testing procedure and will largely document how different tests are carried out, along with the data used for the test, the predicted result, the actual result, and a notes section to add any information that may be of useful.

Testing Report

When testing our solution, we made use of a few different types of test to ensure that our implementation was working to the standards outlined by our requirements. Though we aimed to largely make use of unit tests where possible, there were naturally some cases in which this was not possible, and the appropriate method of testing was selected based on the situation. Due to the limited implementation dictated by the assessment brief, we ran only 12 tests, though all of these tests passes for the final version. This may indicate that perhaps our tests were not exhaustive enough, or that we have not correctly identified requirements that needed to be tested as part of this stage.

Unit Testing

Of the 12 tests we ran, 7 were unit tests, although some of these unit tests included multiple test instances within them. These tests were mainly orientated around testing the unit allocation, as the implementation of this area of the solution was the only one that fully allowed unit testing to be carried out. For tests that used different values but tested the same circumstance, we included the different instances of these tests within the same unit test. We were unable to unit test the map as it relies on GUI input to test.

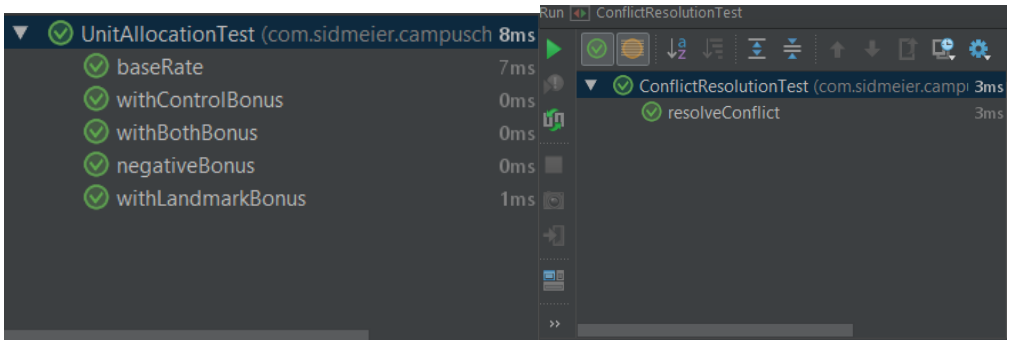

Figure 1 & 2: Unit tests for Unit Allocation & Conflict Resolution respectively

Figures 1 and 2 show the results of running Unit Tests for both the conflict resolution and the unit allocation sections of the program. Due to the way the conflict resolution is implemented, it is impossible for a unit test to test anything other than the function passing out a valid value (True or False), as the random nature of the function means the result cannot reasonably be predicted ahead of time.

Black Box Testing

Due to the nature of the map, we were unable to use unit tests to verify its functionality met the requirements. Instead, we ran the solution as a user would, and used the interface to check that certain conditions were met in order to clarify the extent to which our requirements were met. For all but one of the tests, this involved using the mouse to click, hover, or move around and interact with the map implementation.

Of the 5 tests we conducted for this category, all 5 passed, again potentially highlighting that our test strategy may not be exhaustive enough. A key thing to note is that for tests in this category that had functionality that applies to multiple sectors, only one sector was used in testing. We felt that this was enough to check that the functionality works as intended, as exhaustively testing the entirety of the map is not time efficient. It does however mean that it is a possibility that features tested in this manner may not function correctly on the entire map, though we are confident this is not the case.

Another important point is that for tests 6 & 7, the data for the majority of sector names has not yet been implemented. This was a choice made in the implementation that affords future developers greater flexibility in their approach to the game, as they may choose how sectors will be named and what bonuses should be given to each. Consequently, the tests were only able to be run upon sectors that already had data implemented for them.

Overall we feel our testing is sufficient enough to prove that our solution meets our requirements, in spite of the relatively non exhaustive approach we have taken to them. The scope of the solution leaves limited room for testing, and the number of tests we have carried out reflects that. Unit tests were created for each area of the solution that supported them, and the additional black box testing that encompassed the remainder of the solution was devised directly from the requirements, thereby ensuring that our solution meets them to the fullest extent. This can be seen clearly through our use of test cases and corresponding traceability matrix.

Links

Test Cases: http://www.sidmeiers.me/documents/TestCases.pdf

Test Code: http://www.sidmeiers.me/documents/code.zip

Traceability Matrix: http://www.sidmeiers.me/documents/TraceabilityMatrix.pdf

Citations:

[1] - S. Bechtold et al, “Junit 5 User Guide”, 2018, [Online], Available: http://junit.org/junit5/docs/current/user-guide/