University of York

Department of Computer Science

SEPR - Assessment 2

Testing Report

Team Craig

Thomas Burroughs

Huw Christianson

Joseph Frankish

Isaac Lowe

Beatrix Vincze

Suleman Zaki

Testing Approaches and Methods

After assigning roles for Assessment 2, the three team members assigned the role of 'Software tester' undertook research to select potential software testing methods for our project. This was done before any implementation of our game began, as the group's prior knowledge of game development suggested that testing methods may influence our coding strategy and plan.

One of the testing methods we will be using is automated unit testing, which is an example of white-box testing. Unit testing aims to test program components such as methods and object classes [1]. When testing object classes, we will test all operations associated with the object; set and check values of the attributes of that object and place the object into all its possible states [1]. Our original plan was to test each of the methods associated with the object in insolation, but it became clear that this was not suitable due to the need for sequencings of methods and factors involving inheritance. Due to this a we will also involve test sequencing for some methods. Our unit tests have been simplified by making use of Java's assertion functionality which allows us to test whether an invariant has been preserved or not. On the running or a test suite our tests will be executed automatically is a test flag is set. Careful deliberation was made over choosing the most suitable test cases and decided to follow Somerville's approach to unit test cases. We will have two types of testing. One which shows the normal operation of a component (does it do what is it supposed to?) and another which uses abnormal inputs to test whether the component defects or crashes. Unit testing is suitable for our small project as it allows us to build on previous test from previous iterations. As new components (e.g. methods and functions) are created we can test each of these components in isolation and then test the methods in sequences as the complexity of our code develops through each iteration until the final build is achieved.

Black box testing will also be used for further testing of our code and will be completed at the end of the implementation process. The main aim of black box testing is to validate both our functional and non-functional requirements [2] (shown in our Assessment 1 deliverables). The requirements specification will be used to create test cases for which testers will form an expected outcome before implementation is started. Each of these tests will then either be given a 'Pass' or a 'Fail' depending on whether the testers believe that the actual outcome of the test sufficiently matched the expected outcome. This type of testing requires the testers to have no prior knowledge of the internal code structure of the software. This is suitable for our project as two of the three testers will not be involved in the implementation process. Another justification for this testing type is that a tester does not have to be technical or have good programming skills which helped with assigning roles to members with less programming experience.

Software inspection will also take place during the testing stage. This is particularly useful for incomplete projects [1] as ours will be at the end of Assessment 2. After each iteration or Scrum cycle, we will make software inspection to assess the readability of our code and whether the code created by each programmer complies with the group's code standards. These standards were created by the group before the implementation began and was agreed upon by both the testers and the programmers. We will also use software inspection to identify defects in our code and whether certain blocks of code could be made more efficient, however this is not our main aim with testing method.

A traceability matrix will also be created to show the relationship between our test cases and each of our requirements within the requirements specification. This will not only help to show whether each tests are useful for testing each requirements and to ensure all potential requirements have been tested but also to help with testing in further assessments. As requirements change later in the assessment, we will be able to use our traceability matrix to trace which test cases needs to be altered helping us to abide by our chosen agile methodology.

Our testing methods have to be adjusted in order to follow our agile methodology which at the current time is an adaption of the scrum methodology. Since each of our cycles are only short, testing will need to be done very frequently without a loss of quality. To maximise our time, we will prioritise requirements depending on their risks as seen in the Risk Assessment document. This is necessary as clearly not everything can be tested with such a fast pace approach. The team will also focus on good communication between programmers and testers to reduce the time between writing a block of code and the testing of that block of code. As mentioned previously, the traceability matrix will be used to track the testing of requirements and will to save time by avoiding creating tests for requirements that already have numerous associated test cases.

Block-box Testing Statistics and Results

Two testers responsible for black-box conducted 32 black-box test of which 9 tests failed overall. A failed test is considered to be a test whose outcome did not sufficiently match the expect outcome. The remaining 23 black-box tests passed on the first attempt and required no further action. For most of the tests, the test cases were devised before implementation began, however, suitable changes to test cases were made as and when large changes in architecture and game features were made. For example, the introduction of collectable 'keys' to unlock new map locations required a change in test cases, as a winning state could not be achieved without first having a successful test for collecting keys.

Of the 9 failed black-box tests, 7 tests (**BB3, BB9, BB11, BB12, BB20, BB22, BB23**) failed due to the fact that the feature these tests were testing were not required nor implemented for this assessment. On the devising of the tests, the intention was to implement these additional features but because of time constraints and limited resources. No further action was taken for any of these tests and there was no code for these features to which improvements could be made.

One of the remaining test (**BB26**) failed as a result of miscalculations in map size. Programmers used the test result to implement a suitable offset was placed on the player's position to allow a collision to be detected which prevented the camera from moving passed the map boundary. This rectified the error in the test and a second test was ran (**BB27**) with a slightly stricter expected outcome. This test passed on its first attempt after rectifying previous error. Despite this, the failure of test, did not affect the game's ability to meet the requirements for Assessment 2 so rectifying the failure was not given a high priority.

The final failed test (**BB13**) failed because the result of the test depended on the implementation of another feature which at this time has not been implemented. This test stated that the black-box test should produce an outcome where the player could see the character's health increase by 10. However, because a HUD has not implemented this test failed. As a result of this, another test (**BB13.2**) was created with a different test case which did not require a HUD. This test passed first time. The rectification of this error was made a high priority as it affected whether our game would meet the requirements for this assessment, specifically **REQ ID: F3.** Test **BB13.2** showed that after refactoring this requirement was met.

A suitable link to black-box testing evidence is given at the bottom of this document.

Unit Testing Statistics and Results

Unit testing has been handled by running a test suite before launching the game if a test flag is set. This suite automatically executes a series of actions and makes use of java assertion functionality to make sure that the correct invariant is preserved. All our unit tests passed in the final test run and due to this no further action was required. Due to the nature of our program not all functionality can be unit tested. This is caused by the fact that a lot of functionality only works in the context of events fired by the engine our game runs on top of and we are unable to trigger these events on demand. Because of that the more complex functionality of our project will need to be tested via black box testing as described above. A suitable cross-over and overlap between our black-box tests and Unit tests ensured that all requirements for Assessment 2 have been met as is shown in evidence tables and traceability matrix.

A suitable link to Unit testing evidence is given at the bottom of this document.

URL Links to Testing Materials

Below are links to our evidence for both Black-box testing and Unit Testing. Each test has been assigned a suitable Test ID to allow traceability throughout all our deliverables. The Black-box testing table includes the expected outcome/actual outcome along with a pass/fail status. The Unit testing table includes the category which the test is testing, the name of the test, a description of what the test does along with a pass/fail status.

Black-Box testing Evidence: https://teamcraigzombie.github.io/assets/downloads/Black-BoxTestingEvidence.pdf

Unit Testing Evidence: https://teamcraigzombie.github.io/assets/downloads/UnitTestingEvidence.pdf

Unit Testing code: https://teamcraigzombie.github.io/assets/downloads/GameTest.java

We have also included a link to our traceability matrix to show the relationship between our test cases and each of our requirements within the requirements specification.

Traceability Matrix: https://teamcraigzombie.github.io/assets/downloads/TraceabilityMatrix.pdf

References

[1]- I. Sommerville. *Software Engineering*. Pearson, tenth ed, 2016

[2]- Software Testing Fundamentals**.** *Black Box testing***.** Available at: **http://softwaretestingfundamentals.com/black-box-testing/**