Evaluation and Testing Report

Mozzarella Bytes

Assessment N°4

Kathryn Dale

Daniel Benison

Elizabeth Hodges

Callum Marsden

Emilien Bevierre

Ravinder Dosanjh

Introduction

Both the approach that we took to evaluating and testing our final product revolved around the requirements. In order to test our final product, we created tests based upon the requirements generated through communication with the client, to ensure that all were met. The justification/source for each requirement can be seen in the requirements document. This allowed us to test the quality of our code by ensuring that there was high code coverage as, once all features had been tested and found to work, we could remove 'dead' code from our project (for the purposes of this project, we defined 'quality' code as efficient, functional code that performs the function at hand with no unused, unnecessary lines of code). We also used boundary testing, as mentioned above, to make sure the product works exactly as intended. For our evaluation, we had a physical copy of the requirements in front of us and played the game, commenting on each requirement as it became relevant [1]. Our requirements were developed from our product brief, as well as from regular communication and clarification with the client - hence by using the requirements to evaluate we were able to determine how well the product met our brief. The final requirements can be found here on our website.

Testing

Prior to testing, we decided to use the research and justification from our previous assessment [2] to ensure that we could best test our product, as this worked very well for us before and we are comfortable with how it works. It is not practical, or in many cases possible, to exhaust every possible scenario when testing a developed project. Non-trivial incompatibilities between the LibGDX and JUnit introduced testing by visual inspection – one method adopted by the project. The last testing approach and execution used by the previous team was not done very well. They had limited unit testing, which is partly down to the issues with LibGDX discussed, yet tests that were completed were mostly completed by the group before them. It is also apparent that they removed unit tests for core classes such as Fortress and FireTruck, which are the two of the most critical classes in the game, therefore we thought it was crucial to make sure these classes were fully tested to ensure they worked as intended. We also improved and extended how our tests were documented to be much clearer and more useful. Whereas the previous group used several tables [3] and a confusing Traceability Matrix [4], we utilised the Traceability matrix to its full potential, by ensuring that every one of the requirements had been tested at least once, which is very useful in determining the quality of the whole of our game. We also made sure to include references to our JUnit tests in the Test cases document so that an evaluator can easily locate the code.

End-to-end testing was run frequently on the project, often by immediately implementing methods and testing their functionality. Other end-to-end testing was provided as a complement to unit testing where appropriate, but also for testing features that simply cannot be conducted with automated tests, such as those that require interaction with UI (e.g. T.F.046 testing the main menu) or simply to see some information or visual element (e.g. T.F.045 testing viewing of controls screen).

For Assessment 4, our requirements changed within the project and a new series of tests needed to be written that exercised the resulting features - which included unit testing and end-to-end testing to verify that these features are fully covered. One such feature was the power up. Each power up had a unique effect on the player, therefore we created automated tests to verify that these effects operated as expected. For example, to test that the "Heart" Power up healed the Fire Truck to 100% HP, we created test T.U.013a to

validate this, which records that the HP of the Fire Truck went from 5 to 240 (which is the maximum health of that Fire Truck). Where it was not feasible to do automated tests due to dependency issues with LibGDX and JUnit, we ran end-to-end tests such as T.U.013e, which made sure the Freeze power up reset the freeze cooldown. These tests can be seen in our Test Cases document.

Detailed documentation, such as Test Cases, Traceability Matrix, Test Coverage and Test Results can be found on the project website:

https://db1218.github.io/KroyAssessment4/assessment4/overflow#testing

We documented our tests using Test cases. As our project, and therefore testing, is built around satisfying requirements, we split test cases up into three sheets, one for each type of requirement (User, Functional and Non-functional). Then for each test, we included; ID, description, test category, relevant requirements, code reference, input data / instructions, expected and actual outcome, status and severity. This consistent documentation allows a viewer to quickly and easily see all of our tests.

We created JUnit tests for core classes of the game; *FireTruck*, *Fortress*, *FireStation*, *Patrol*, *BossPatrol*, *Particle*, *PowerUp and Dancer*, as these classes had few dependencies and are important classes for our game.

Code Coverage for Individual Test classes

Class Tested	Method Coverage	Line Coverage
BossPatrol	85%	82%
FireStation	75%	79%
FireTruck	72%	75%
Fortress	52%	58%
Particle	57%	80%
Patrol	55%	65%
PowerUp (package)	45%	56%
Dancer	78%	85%

Code Coverage gave us a good indication as to how much of these classes we have covered in Unit Testing. However, this does not give a complete picture as many of the methods not covered are either related to *rendering* and cannot be unit tested or are *getters* / *setters* that do not need to be tested. While writing unit tests, we encountered issues with initializing objects with heavy dependencies on others. Therefore, where appropriate, we used the Mockito framework to replicate crucial methods from said classes so that the unit tests could function properly. This meant that many of our classes could not be tested, such as LibGDX's *Screens*, we therefore mainly tested entities. Each method that remained untested is listed on our website which should clarify why we deemed it unnecessary or simply impossible to do so.

All 118 JUnit tests passed and can be accessed on our website. For these tests to work, we used Mockito to mock objects that the class we were testing depended on to run. For example, we had to mock *GameScreen* in order to test *FireTruck*, as when playing the game, *FireTruck* needs to be able to access the Tiled map to check whether it can take a valid move, which is accessed in a method in *GameScreen*. As the Tiled Map's functionality is a part of LibGDX, it is not within our responsibility to create unit tests for them (instead we can confirm it works as intended by using the library), therefore we could mock *GameScreen* and pre-empt the values its methods would return, in order to solely test *FireTruck*.

For our Unit tests that involved numerical values, we designed our tests using boundary testing, which was adapted to player scenarios gathered from user feedback and which demonstrated multiple behaviours of one function. For example, to demonstrate a *Fortress'* attacking range (T.F.031) of 8 tiles, we had three JUnit tests, labelled as (T.F.031a, T.F.031b, T.F.031c). '*a*' was to check that the Fortress could hit the Fire Truck when just within range (7.9 tiles away), '*b*' was to check that the Fortress could hit the Fire Truck when it was on the maximum range (8 tiles away), and finally '*c*' was to check that the Fortress could *not* hit the Fire Truck when just out of range (8.1 tiles away). We think these are important to test as they prove that this feature of the game works *exactly* as intended.

The only requirement we did not test was "UR_SCALABILITY", which created the possibility for the game to be played on other platforms. We fulfilled this requirement by using LibGDX as we knew that this framework allowed the game to be run on android. However, we did not test that the game could actually be ported to android or other platforms, hence why the test failed. On the other hand, our game relies mostly on mouse control through dragging and clicking, therefore one could argue it would be straightforward to scale the game by making it run on mobile which primarily relies on touch input. Due to this and the fact that this was not a core requirement, we deem this failure as minor in contributing to the in-completeness of our testing.

Subjective requirements such as; UR_PLAYABLE, UR_ENJOYABILITY, UR_PLAYABLE and UR_MINI_GAME_THEME were hard to test as the outcome of the test is a matter of opinion. We conducted the tests by asking friends and family what they thought and got our results that way. In an ideal situation, we would have asked independent volunteers to test our game, however the Covid-19 lockdown made this impractical. In future to make this test more reliable, we would do it on a much larger scale, with a controlled sample group, as friends and family may have offered a biased response.

Testing can never be complete [5]. It is not possible to test all possible inputs, all game logic, all possible paths, and failures due to user interface decisions or insufficiently detailed requirements. Even though testing cannot be 'complete', it can be thoroughly done. We have evaluated how well our tests validify the quality of our program by comparing them to the final requirements of the product generated and evolved through continuous communication with the client.

In an effort to ensure tests for the project are effective at ensuring it meets such requirements, there exists a link to our *Traceability Matrix* on our web site to assist in determining the completeness of the testing performed. This ensures that for each *User*, *Functional* and *Non-functional* requirement there exists at least one test. This does not, however, eliminate the possibility of insufficient testing, but instead shows that the final requirements have been met.

Evaluation

Our evaluation was done by analysing the requirements in depth and playing the game while actively looking to ensure each requirement was met. This evaluation highlighted to us that there were some issues when playing the game, for example, the map was not fully visible on all devices and trying to learn what each entity did by clicking on it during game play was difficult in the limited time. We were able to improve upon many issues that we detected from this method - the map was edited to make all fortresses visible to users across all devices and the User Manual was updated to include the new features.

Another approach we took to evaluation was to have friends and family play the game and give us feedback. As discussed in Testing above, ideally we'd have had a larger, independent control group to test our game but external factors prevented this. We learnt that the minigame was very difficult for first time users to get to grips with - however with practice this became easier. Part of the reason why the game was so hard to complete on the first few attempts was that the box in which you are supposed to click on the arrow keys at the correct moment was only nominal, and so the system appeared to be unresponsive and would mark you as late or wrong when there's a line of consecutive arrows. This was quite frustrating for users and so we made the game easier to play by giving a greater tolerance for mistakes, making it more responsive and so more enjoyable.

It was very challenging to evaluate whether or not some requirements had been met (e.g. UR_ENJOYABILITY and UR_PLAYABLE) as they are subjective to each player. This method of testing and evaluating is discussed further in the Testing Report above, but in order to try and evaluate these requirements, we had a selection of friends and family play the game and score it out of 10 for enjoyability, and also to rank how difficult they found each level of the game. The average score for enjoyability was 8 and the difficulty scores for easy, medium and hard were 3, 7 and 9 respectively. This demonstrates that we were able to meet the client criteria that the game must be enjoyable to play relatively well. Furthermore, we were satisfied by the volunteer feedback that each difficulty level was sufficiently harder from the last without it being too big a jump in how hard it is to win. One comment we had regarding difficulty was that there was no considerable increase in difficulty for the minigame.

Bibliography

[1] K. Dale and D. Benison, *Requirements Evaluation.* 2020 https://db1218.github.io/KroyAssessment4/assessment4/overflow/Requirements%20Evaluation.pdf

[2] D. Benison, *Software Testing Research Report.* 2020

[3] H. Vajja, *Automated Test Cases.* 2020 https://db1218.github.io/KroyAssessment4/assessment3/overflow/automatedCases.pdf

[4] H. Vajja, *Traceability Matrix.* 2020 https://db1218.github.io/KroyAssessment4/assessment3/overflow/traceabilitymatrix.pdf

[5] C. Kaner and R. L. Fiedler, *Foundations of Software Testing*. Context-Driven Press, 2013.